International Workshop on UNIX - Based Software Development Environments

Co-sponsored by:

Japan UNIX Society and SIGMA Project (Japan)
USENIX Association (USA)

January 16-18, 1991
Dallas, Texas

Table of Contents

International Workshop on UNIX-Based Software Development Environments

January 16 - 18, 1991
Dallas, Texas

Emeraude, a UNIX-based Implementation of PCTE
Regis Minot

NeTS: Computer Network Analysis Tool
Hideo Nakano

Experiences of a Transition to SunOS Development under the NSE
Tadd Ottman

WhiteCap - A C/C++ Development Environment
David R. Reed

NGCR Project Support Environment Standards
Carl Schmiedekamp

C Development on PCTE
Ian Simmonds

Supporting Parallel Programming Environments with Shared Persistent Data Structures
David C. Sowell, Karsten Schwan

Integrated Project Support Environments: Benefits and Function
Walter Van Riel

Program Co-Chairs

Noboru Akima
Sigma Project
5th Akihabara Sanwa Bank Building
3-16-8 Soto-Kanda, Chiyoda-Ku
Tokyo, Japan 101

Stuart Feldman
Bellcore
445 South Street
Morristown, NJ 07962-1910
USA

WELCOME TO THE WORKSHOP

Stuart Feldman

Bellcore
Morristown, New Jersey, USA

Welcome to the International Workshop on Unix-Based Software Development Environments. We hope you will this meeting interesting and valuable. The presentations are expected to set the tone for the meeting and to seed the discussions of the papers and of related topics. We hope this will be a highly interactive meeting, with much sharing of experience, techniques, tricks of the trade, gripes and compliments.

Attendees and presentations represent a wide variety of organizations, backgrounds, and locations, with contributions from America, Europe, and Asia. One of the original motivations of this meeting was to make information on the Japanese Sigma project available to others, and to share experiences with that group, so I am delighted to welcome the people associated with that major effort, and glad we have been able to schedule four presentations relating to it.

This book contains the position papers that were received in acceptably camera-ready form by the deadline. Others will be handed out at the workshop. During the meeting we will discuss the possibility of producing an archival proceedings volume with more detailed papers.

Best wishes for a fine meeting!

THE BELLCORE ADVANCED SOFTWARE ENVIRONMENT (BASE)

Maryam Asdjodi
Chit F. Chung
Victor S. Gregg
Reva Y. Leung
Robert D. Mitchell
Todd S. Moyer
Josh Nabozny
Abe Shliferstein
Suresh Subramanian
Gomer Thomas

ABSTRACT

Bellcore has an excellent record of producing quality software. The Bellcore Advanced Software Environment (BASE) organization has been chartered with the design, implementation and deployment of an integrated software environment that can further improve our ability to develop software. The environment will be information-centered, tool-rich, user-attractive and cost-effective. BASE will specifically effect several quality enhancements that will benefit both people and processes, leading to improved responsiveness to client's needs and better quality of Bellcore deliverables.

1. INTRODUCTION

The Bellcore Advanced Software Environment (BASE) organization has been chartered with the design, implementation and deployment of an integrated software environment for developers, planners, testers, managers, and support staff in the Software Technology and Systems (ST&S) area of Bellcore. The overall BASE goal is to better quality and improve overall productivity by deploying an integrated platform for software development [1].

BASE will promote quality enhancements in the workplace by providing automated assistance for several manual procedures, and by reducing paper flow in an organization. A major theme is to foster tool, information, and process reuse by ensuring ubiquitous access to the right thing in the right form at the right time. The environment is expected to benefit both people and processes: staff will be educated in and provided access to modern software engineering facilities and practices, while processes will benefit from better tools and communication facilities. The end result will be improved responsiveness to client's needs and better quality of Bellcore deliverables.

We describe the strategies which are instrumental in realizing BASE on a standard workstation platform. Specifically, the ways by which software technology can realize quality improvements will be presented.

2. BASE REQUIREMENTS

The software industry is constantly undergoing technological change. Software systems are becoming increasingly complex as users demand more functionality, friendlier user interfaces, and an ability to operate in distributed networks. In many ways, the ST&S area of Bellcore faces additional challenges, because the development of large telecommunications operations systems must satisfy a number of diverse customer needs in a variety of operating environments.

Fundamentally, BASE is attacking the communication problem that is the bottleneck in large-scale software development. People often do not have access to the information or tools that they should or could have, so they resort to manual techniques and the expertise of their colleagues to get work done. Often, even if the information is available, it is in paper form. This makes it difficult for users to access and relate software development artifacts ranging from source code and English text to design diagrams and pictures in user manuals.

The BASE solution tries to migrate knowledge that is usually tied to human beings into the computing environment that is actually used for software development. We believe that the increasing information needs in the software development process make it incumbent to make this information widely available. An integrated environment can ameliorate many tedious chores in the workplace, and can foster use and reuse in an organization.

3. BASE INTEGRATION STRATEGIES

The BASE organization is addressing the impediments described above by exploring a shared platform for software development. This platform shields the user from disparate user interfaces, toolsets and data formats by providing a seamless software engineering environment. The key technical concept in the BASE solution is integration - at the user interface level, at the process level and at the information level [2].

3.1 User Interface Integration

User interface or presentation integration focuses on providing a single facility for access to and information about an advanced toolset and its associated data. This facility will also provide context-sensitive help that is tailorable based on a user's familiarity with the environment. The net result of this flavor of integration is that BASE users have uniform access to a set of standard tools they may need. Of course, they can customize this collection of tools, subject to organizational policy (e.g., they can pick their editor of choice, but may be required to use a specific publishing package).

3.2 Process Integration

Process integration tries to assist people in getting their work done. This facility guides users in fulfilling tasks associated with particular roles (e.g., manager, developer, system tester) and also monitors user activities. This facilitates process management and process coordination, and also enables the environment to do some intelligent background processing (e.g., when a system passes the development phase, the relevant system testers can automatically be notified). An interesting aspect of process integration is the ability to automatically invoke appropriate tools with appropriate templates (e.g., when a design document is to be created, the correct design tool is brought up, together with the correct design template for the organization).

Process integration is especially beneficial in educating staff about corporate practices and standards, and in ensuring traceability. This will improve responsiveness to client needs. The process guidance mechanisms and the templates also serve to make process information ubiquitous, and hence, reusable. This should help staff get on board quickly when moving between different organizations. Thus, one of the benefits that BASE can provide is to support Bellcore standards so they can be applied more uniformly and with decreased effort.

It should be stressed that the process integration facility is not rigid. Although certain corporate-level processes will be standardized, there are mechanisms to customize the individual subprocesses.

3.3 Information Integration

Information integration is one of the most critical parts of the BASE environment [3]. It refers to a unified and shared information management capability that allows users, be they people or processes, to get the right information, at the right place, at the right time, in the right form. This means that users can access information on-line without worrying about the disparities in data formats between tools. An excellent example of this will be the ability for developers to simultaneously have access (via multiple windows) to both the source code module they are working on, and the design document that explains the change they should be making to the code.

An important aspect of information management is the ability to track and maintain relations between information objects. This implies that related pieces of data that are often housed in separate places can be linked together. Again, the end result should be for users to get information in on-line form quickly.

4. TOOL QUALITY INITIATIVES

In order to attract users and get the greatest benefits, BASE must provide a rich selection of tools. The integration facilities should make these tools broadly useful and attractive. BASE will supply an integrated collection of tools such as communications tools (e.g., electronic mail), documentation tools, administration tools, project management tools and general reuse tools (e.g., to do efficient searches through libraries of design documents). In addition, the specific tools required by the various people in a software development effort need to be accessible. Although there is a good supply of these tools in actual use, there are newer and more powerful tools that can make developers even more effective. These tools range from language-specific editors to advanced analysis tools to structured requirements, design, and testing tools [4]. The main goal of this tool effort is to define a standard set of tools and to provide integrated access to these tools.

5. CONCLUSION

We expect ST&S, Bellcore and our clients to benefit from the widespread use of BASE. The main result will be an improved ability in ST&S to produce and support quality software. There are currently various subcultures, each with their own tools, history, and computational approaches. A widely deployed integrated environment would make it possible to share the best tools across the entire organization, and to bring in new technology from outside in a timely fashion. On-line access to a myriad of information will also reduce paper flow and improve the

timeliness and quality of work products. BASE will require changes to some old ways of doing things and introduce new approaches as well. The net goal is to increase the shareability of resources, thereby fostering reuse and naturally improving quality and productivity of the people and processes involved in Bellcore software development.

6. REFERENCES

[1] Maryam Asdjodi, Stuart I. Feldman, Nicholas J. Filippis, Kurt A. Gluck, Reva Y. Leung, Marc F. Pucci, Doron Shalmon and Suresh Subramanian, *Recommendations of the Joint ST&S-ARA Programming Environment Task Force,* Bell Communications Research Internal Document, June 1989

[2] Victor S. Gregg, Reva Y. Leung, Robert D. Mitchell, Todd S. Moyer and Suresh Subramanian, *Bellcore Advanced Software Environment (BASE) -- Integration Strategies,* Bell Communications Research Internal Document, June 1990

[3] Josh Nabozny and Gomer Thomas, *Bellcore Advanced Software Environment (BASE) -- Information Management,* Bell Communications Research Internal Document, May 1990

[4] Maryam Asdjodi and Chit F. Chung, *Bellcore Advanced Software Environment (BASE) -- Tool Set and Tool Integration,* Bell Communications Research Internal Document, August 1990

MARVEL 2.6
A Unix-Based Software Development Environment Kernel

Position Paper

Naser S. Barghouti* Gail E. Kaiser†
Columbia University
Department of Computer Science
New York, NY 10027
naser@cs.columbia.edu, (212) 854-8182
kaiser@cs.columbia.edu (212) 854-3856

July 25, 1990

*Barghouti is supported in part by the Center for Telecommunications Research.

†Kaiser is supported by National Science Foundation grants CDA-8920080, CCR-8858029 and CCR-8802741, by grants from AT&T, BNR, Citicorp, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

Introduction

As software becomes larger and more complex, the need for assistance in the development of software projects becomes more acute. Software development environments (SDEs) aim to satisfy this need by monitoring the software development process, automating parts of it, and/or recognizing a specific development process in order to assist the developer in planning what to do next. SDEs generate and manipulate large amounts of data in the form of source code, object code, documentation, test suites, etc. It is often the case that SDEs depend on the operating system platform, on which they are implemented, for storing and organizing data as well as carrying out development operations. We briefly describe the MARVEL Unix-based SDE [2], and discuss how Unix has affected, both positively and negatively, the design and implementation of MARVEL.

Key Concepts in MARVEL

The long-term goal of the MARVEL project is to develop a kernel for multi-user development environments that use knowledge about the software development process of large-scale projects to support the organizational and automation needs of multiple developers cooperating on these projects. MARVEL uses a rule-based specification to model the software development process and an object-oriented specification to model the organization of the project's data. These specifications, which are written in the MARVEL Strategy Language (MSL), provide the basis for automating parts of the development process. We envision that libraries of MSL specifications will be built, maintained and shared by project administrators. The MARVEL kernel has facilities to `load` MSL specifications to produce a target MARVEL environment that understands how to organize the project's data (i.e., the *data model*), and what kinds of operations will be performed on this data and when to invoke these operations automatically (i.e., the *process model*).

The data model is specified in terms of classes, each of which consists of a set of typed attributes that can be inherited from multiple superclasses. Attribute types include simple types (integer, string, etc.), files, sets and directed links. Set attributes contain instances of other classes as their values, thus implementing composite objects, and giving the MARVEL object management system (OMS) a hierarchical traversal capability. Links can be generic, or point to specific attribute types or classes, thus giving the MARVEL OMS arbitrary graph traversal capability. The MARVEL OMS supports creation and deletion of objects according to the data model. Existing software systems can be immigrated into MARVEL using the Marvelizer tool.

The process model is described in terms of rules that specify the behavior of the tailored MARVEL environment in terms of what commands are available and what kind of automation the environment should provide. MARVEL supports a model of automation called *opportunistic processing*, which employs backward and forward chaining among rules to automatically initiate activities, and thus relieve the developers from mundane chores. The set of rules that are loaded into a MARVEL environment form a network of possible forward and backward chains. Unlike the **make** tool, the firing of MARVEL's rules is not based on timestamp comparison, but on the satisfaction of powerful logical conditions (explained later).

In order to facilitate the management and manipulation of software objects, MARVEL provides an X11-based graphics interface (including a full browser) and a corresponding command line interface for batch processing and dumb terminals. The graphics interface includes a rule debugging facility that helps the user visualize and step through the execution of rule chains. We have developed MARVEL environments for programming in C and C++, writing documents using a text editor and the Scribe text formatter, and maintaining a simple library catalog system.

We have completed the implementation of a single-user version of the kernel, MARVEL 2.6 [1], which runs under several variants of the Unix operating system, including SunOS 4.0.3 on Sun3s and Sun4s, Ultrix 3.1 on DecStation 3100s, AIX 2.2.1 on IBM RTs, and HP-UX on HP 9000s. The code is very portable, so we anticipate few problems in running MARVEL 2.6 on other Unix-based hardware/software platforms[1].

Unix as an Implementation Framework

Implementing MARVEL on top of Unix has enabled us to exploit well-known Unix features such as the ability to invoke arbitrary tools from within programs, and the ease of access and manipulation of the hierarchical file system. There is a rich set of software development tools, such as system configuration management tools (e.g., `make` and `rcs`), that have been implemented on top of Unix; such tools can readily be made available from within an SDE to complement the other assistance provided to the developers. The Unix file system provides binary and text files, as well as a hierarchical directory structure, in which different parts of the software project are stored. The file system presents a common interface to tools.

These Unix features have simplified the implementation of two key concepts in MARVEL: (1) providing a persistent objectbase manipulated by the OMS and immigrating existing data to a MARVEL objectbase; and (2) supporting commercial off-the-shelf (COTS) tools.

Providing a Persistent Objectbase

MARVEL provides persistence of objects through an objectbase that combines the Unix file system and an in-memory data structure. Objects in a particular MARVEL environment are instances of the classes defined in the project's data model. As described earlier, classes have attributes that are inherited by objects. Attributes are of three different categories: (1) status attributes, which are of simple types; (2) data attributes, which are binary and text files containing data belonging to the object; and (3) structural attributes such as sets, which provide the ability to create composite objects, and links, which describe arbitrary relationships between objects. MARVEL stores all objects belonging to a project in a "hidden" file system rooted at a designated directory. The structural and data attributes of objects are mapped directly onto directories, subdirectories and files. For example, a composite object containing subobjects is mapped to a directory and each subobject is stored as a

[1] We are distributing MARVEL 2.6 to interested educational institutions and industrial sponsors. The distribution includes a tape, on one of several types of media, and over 350 pages of documentation, including a user's manual and an implementor's manual. Contact Israel Ben-Shaul, (212) 854-2736, israel@cs.columbia.edu for licensing information.

subdirectory of that directory. Status attributes of all objects are stored and manipulated in an in-memory data structure, which is checkpointed to the file system before every significant update operation.

MARVEL provides two tools, Marvelizer and Organ, to immigrate existing software systems whose data is stored directly in the Unix file system into a MARVEL objectbase and to reorganize the components of software systems within an objectbase, respectively [3]. Since the mapping between a MARVEL objectbase and the Unix file system is straightforward, immigrating existing data has been greatly simplified. The code of the MARVEL system itself has been Marvelized into a MARVEL objectbase; this takes about twenty minutes elapsed time on a Sun 3/60 with a swap disk.

Supporting External Tools

Since there are numerous COTS tools available for developers to use in developing their projects, it is essential that an SDE provide the capability of conveniently accessing these tools. The SDE can integrate these tools if it provides mechanisms for coordinating the invocation of tools and synchronizing their access to shared data. MARVEL achieves this through its rule-based process model. MARVEL rules are more complicated then their expert systems ancestors because of the need to support external tools. Each rule consists of a *precondition* that must be true for the rule to fire; an *activity*, which is a general mechanism to execute arbitrary external tools; and multiple *postconditions*, each of which is an assertion of the effects of executing the external tool as they are reflected on the in-memory portion of the objectbase. When the activity is completed, exactly one of the postconditions is asserted, depending on the actual results of the activity. Multiple postconditions are necessary in order to treat the activity as a "black box", whose outputs are specified by the postconditions. For example, a compiler may either succeed, producing object code, or fail, returning error messages. It is impossible to determine which one of these situations will occur without executing the compiler.

The "black box" nature of MARVEL activities is implemented by embedding the invocation of an external tool within an *envelope*, which is responsible for translating data from the MARVEL representation to one that is accepted by the external tool and conveying the results of invoking the tool back to MARVEL. This mechanism has enabled MARVEL to support COTS tools executing on the file system, where modification of a tool is impossible (or at least avoided) and it is difficult to directly monitor effects. Unix has helped us in implementing this support by providing shell languages that can be used to write envelopes. Currently, we code envelopes as shell scripts and allow these scripts to extract project data needed by the external tool. In the case of a compiler, the shell script would fetch MARVEL objects representing the source code that needs to be compiled, re-arrange these objects in a format acceptable to the compiler (e.g., merge several files representing one module object into a single text file), and run the compiler on these objects. The result (i.e., success or failure) is recorded in the **status** shell variable, which MARVEL reads and depending on its value, asserts one of the postconditions.

There is a significant limitation, however, because of the restrictions that Unix imposes on the communication between a program and a shell script. A shell script can record the

effects of a tool by either assigning values to shell variables, or storing results in text files. It would be desirable for MARVEL to be able to pass complete objects, while maintaining their structure, to an envelope, which would execute an external tool, record the results in the attributes of the objects, and return the modified objects to MARVEL's rule processor. This would allow us to protect the objectbase from being directly accessed by envelopes, which currently must understand the internal format of the objectbase. Unix does not readily provide mechanisms to implement such support.

Future Development

We have already begun the next major phase of the MARVEL project, which is the implementation of a client/server model for the object management system, moving the system away from it's single-user limitation. The client/server model will provide the necessary support for the implementation of a multi-agent model, which involves support for cooperative and long transactions, and synchronization of multiple intelligent agents (*agents* include both human users and rules). It will be interesting to see how Unix will influence the future development of MARVEL . We expect to face some problems because of the lack of support for multiple threads of execution within a Unix process and the lack of shared memory among user processes, which might simplify the implementation of cooperative transactions.

References

[1] G. E. Kaiser, N. S. Barghouti, and M. H. Sokolsky. Preliminary experience with process modeling in the marvel software development environment kernel. In *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, Kona HI, January 1990.

[2] G. E. Kaiser, P. H. Feiler, and S. S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[3] M. H. Sokolsky. Data migration in an object-oriented software development environment. Master's thesis, Columbia University Department of Computer Science, April 1989, Technical Report CUCS-424-89.

Experience Using CVS: The Concurrent Versions System

Brian Berliner

Sun Microsystems, Inc.
Rocky Mountain Technology Center
5465 Mark Dabling Blvd.
Colorado Springs, CO 80918
berliner@sun.com

ABSTRACT

This extended outline is being submitted for review for inclusion at the USENIX Software Development Environments Workshop, January 1991. The Concurrent Versions System (**cvs**) is briefly described. Reflections are made on experience using **cvs** — its plusses and minuses, its performance, its interaction with UNIX, and its availability.

Note that this extended outline is very rough and the final product will likely have sections that are extremely expanded or extremely compressed in size and content.

1. What is CVS?

1.1. Description

cvs (Concurrent Versions System) is a front end to the RCS [Tichy] revision control system which extends the notion of revision control from a collection of files in a single directory to a hierarchical collection of directories each containing revision controlled files. Directories and files in the **cvs** system can be combined together in many ways to form a software release. **cvs** provides the functions necessary to manage these software releases and to control the concurrent editing of source files among multiple software developers. **cvs** is intended to be run within a UNIX[1] environment. The six major features of **cvs** are listed below, and will be described in more detail in the following sections:

1. Concurrent access and conflict-resolution algorithms to guarantee that source changes are not "lost."
2. Support for tracking third-party vendor source distributions while maintaining the local modifications made to those sources.
3. A flexible module database that provides a symbolic mapping of names to components of a larger software distribution. This symbolic mapping provides for location independence within the software release and, for example, allows one to check out a copy of the "diff" program without ever knowing that the sources to "diff" actually reside in the "bin/diff" directory.

[1] UNIX is a registered trademark of AT&T.

4. Configurable logging support allows all "committed" source file changes to be logged using an arbitrary program to save the log messages in a file, notesfile, or news database.
5. A software release can be symbolically tagged and checked out at any time based on that tag. An exact copy of a previous software release can be checked out at any time, *regardless* of whether files or directories have been added/removed from the "current" software release. As well, a "date" can be used to check out the *exact* version of the software release as of the specified date.
6. A "patch" format file [Wall] can be produced between two software releases, even if the releases span multiple directories.

The sources maintained by **cvs** are kept within a single directory hierarchy known as the "source repository." This "source repository" holds the actual RCS ",v" files directly, as well as a special per-repository directory (CVSROOT.adm) which contains a small number of administrative files that describe the repository and how it can be accessed. See Figure 1 for a picture of the **cvs** tree.

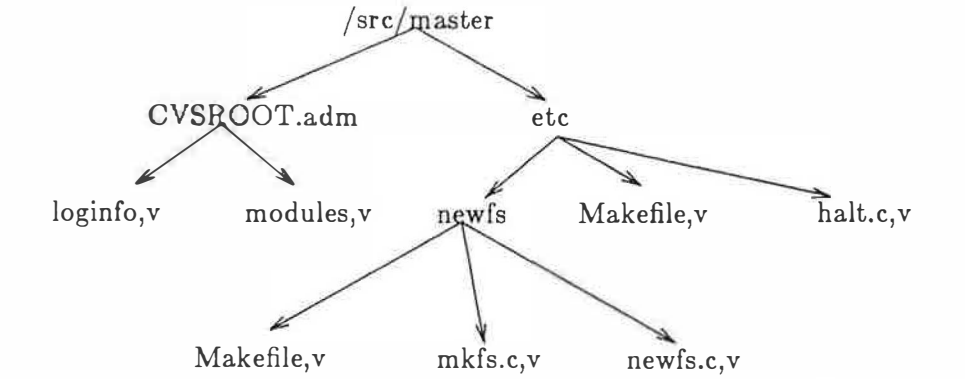


Figure 1.
cvs Source Repository

1.1.1. CVS Propogates Files

It is not the goal of **cvs** to solve any and all CASE problems with a single tool. That would make the tool too monolithic. **cvs** is essentially a source server, and takes care to manage revisions of files that relate to releases of software. It is up to software that can be layered on top of **cvs** to solve other, harder, issues of software configuration management.

1.1.2. Layered Approach for Fancier Features

By taking a layered approach to software configuration management, the problems that **cvs** must solve and limitted (and doable). This approach puts a particular level of abstraction within **cvs**, but allows other levels to be added on top to form a more complex system. This allows each site that uses **cvs** to establish their own policy of release control procedures without being constrained by the underlying tools (**cvs** and RCS).

1.2. Major Features of CVS

1.2.1. Software Conflict Resolution[2]

cvs allows several software developers to edit personal copies of a revision controlled file concurrently. The revision number of each checked out file is maintained independently for each user, and **cvs** forces the checked out file to be current with the "head" revision before it can be "committed" as a permanent change. A checked out file is brought up-to-date with the "head" revision using the "update" command of **cvs**. This command compares the "head" revision number with that of the user's file and performs an RCS merge operation if they are not the same. The result of the merge is a file that contains the user's modifications and those modifications that were "committed" after the user checked out his version of the file (as well as a backup copy of the user's original file). **cvs** points out any conflicts during the merge. It is the user's responsibility to resolve these conflicts and to "commit" his/her changes when ready.

Although the **cvs** conflict-resolution algorithm was defined in 1986, it is remarkably similar to the "Copy-Modify-Merge" scenario included with NSE[3] and described in [Honda] and [Courington]. The following explanation from [Honda] also applies to **cvs**:

> Simply stated, a developer copies an object without locking it, modifies the copy, and then merges the modified copy with the original. This paradigm allows developers to work in isolation from one another since changes are made to copies of objects. Because locks are not used, development is not serialized and can proceed in parallel. Developers, however, must merge objects after the changes have been made. In particular, a developer must resolve conflicts when the same object has been modified by someone else.

In practice, I have found that conflicts that occur when the same object has been modified by someone else are quite rare. When they do happen, the changes made by the other developer are usually easily resolved. This practical use has shown that the "Copy-Modify-Merge" paradigm is a correct and useful one.

1.2.2. Tracking Third-Party Source Distributions

Currently, a large amount of software is based on source distributions from a third-party distributor. It is often the case that local modifications are to be made to this distribution, *and* that the vendor's future releases should be tracked. Rolling your local modifications forward into the new vendor release is a time-consuming task, but **cvs** can ease this burden somewhat. The **checkin** program of **cvs** initially sets up a source repository by integrating the source modules directly from the vendor's release, preserving the directory hierarchy of the vendor's distribution. The branch support of RCS is used to build this vendor release as a branch of the main RCS trunk. Figure 2 shows how the "head" tracks a sample vendor branch when no local modifications have been made to the file. Once this is done, developers can check out files and make local changes to the vendor's source distribution. These local changes form a new branch to the tree which is then used as the source for future check outs. Figure 3 shows how the "head" moves to the main RCS trunk when a local modification is made.

---

[2] The basic conflict-resolution algorithms used in the **cvs** program find their roots in the original work done by Dick Grune at Vrije Universiteit in Amsterdam and posted to **comp.sources.unix** in the volume 6 release sometime in 1986. This original version of **cvs** was a collection of shell scripts that combined to form a front end to the RCS programs.

[3] NSE is the Network Software Environment, a product of Sun Microsystems, Inc.

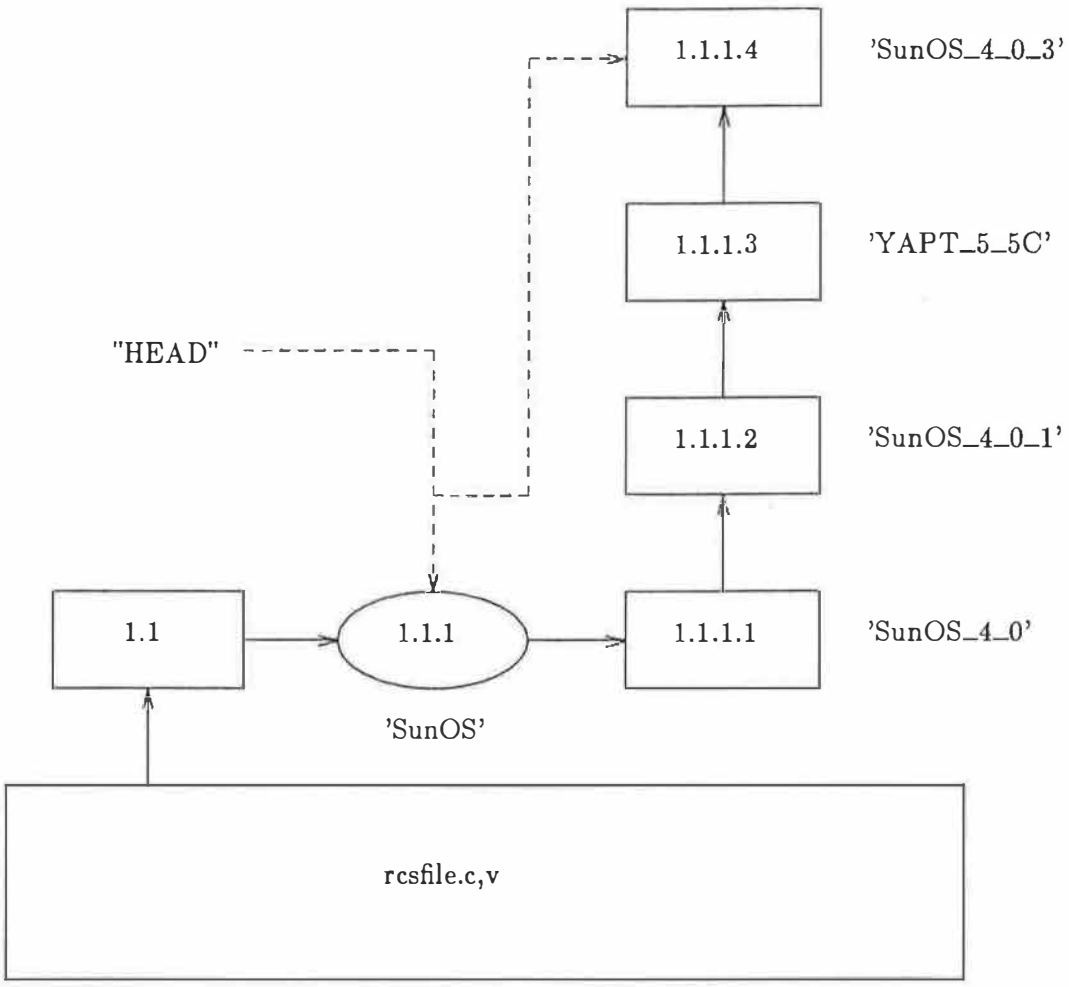


Figure 2.
cvs Vendor Branch Example

When a new version of the vendor's source distribution arrives, the **checkin** program adds the new and changed vendor's files to the already existing source repository. For files that have not been changed locally, the new file from the vendor becomes the current "head" revision. For files that have been modified locally, **checkin** warns that the file must be merged with the new vendor release. The **cvs** "join" command is a useful tool that aids this process by performing the necessary RCS merge, as is done above when performing an "update."

There is also limited support for "dual" derivations for source files. See Figure 4 for a sample dual-derived file. This example tracks the SunOS distribution but includes major changes from Berkeley. These BSD files are saved directly in the RCS file off a new branch.

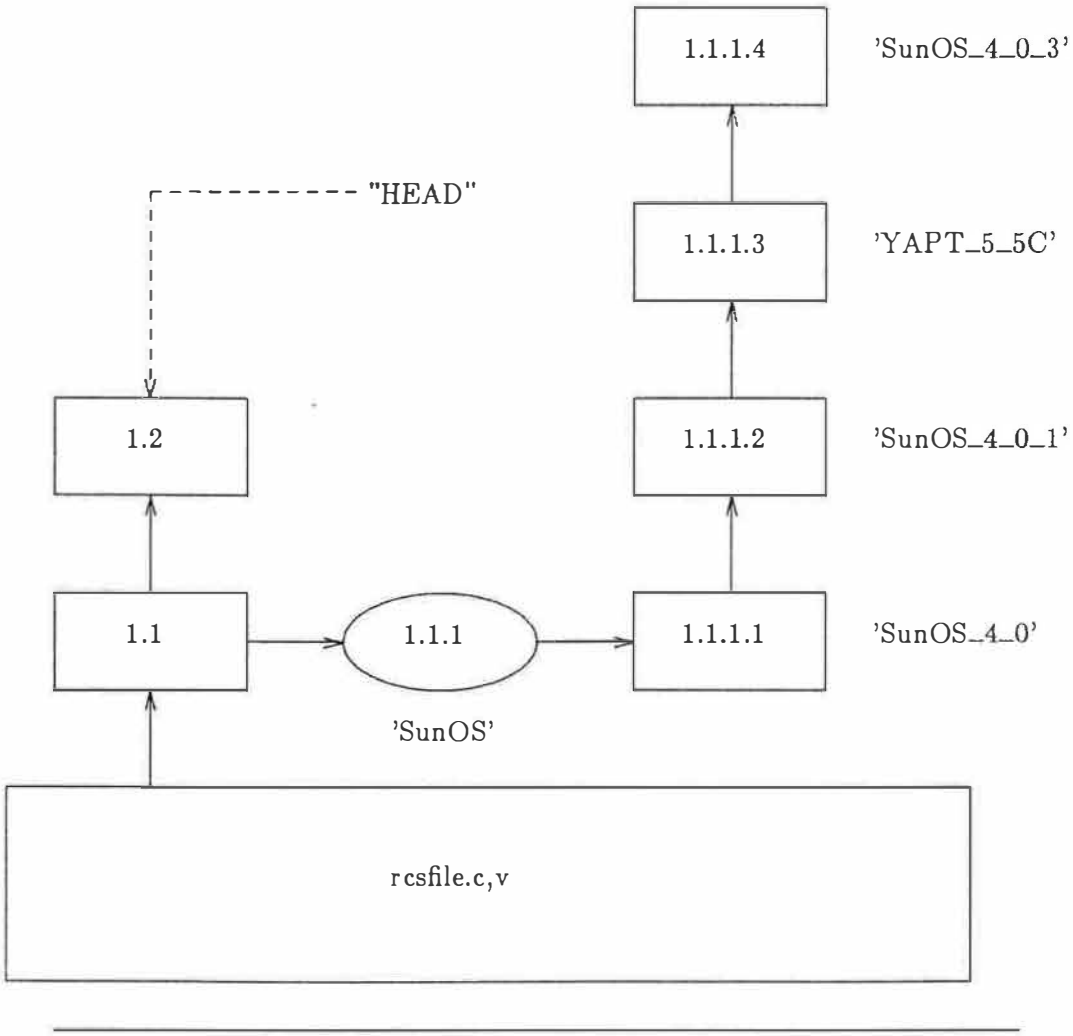


Figure 3.
cvs Local Modification to Vendor Branch

1.2.3. Location Independent Module Database

cvs contains support for a simple, yet powerful, "module" database. For reasons of efficiency, this database is stored in **ndbm**(3) format. The module database is used to apply names to collections of directories and files as a matter of convenience for checking out pieces of a large software distribution. The database records the physical location of the sources as a form of information hiding, allowing one to check out whole directory hierarchies or individual files without regard for their actual location within the global source distribution.

Consider the following small sample of a module database, which must be tailored manually to each specific source repository environment:

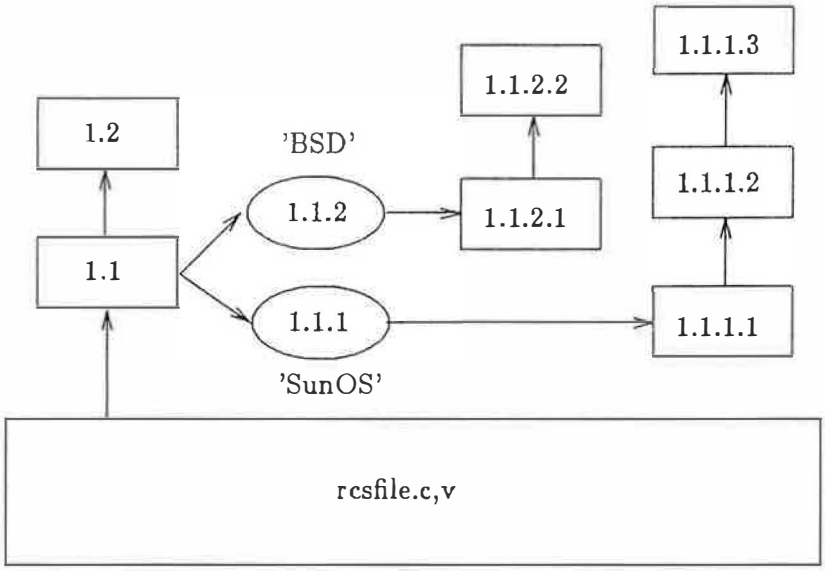


Figure 4.
cvs Support For "Dual" Derivations

The PMS Version Base
(July 1990 — Draft)

C. Breeus, L. Claeys*, M. Lacroix** and J.E. Waroquier**

Center for Software Technology* & Philips Research Laboratory**
Av. Abert Einstein, 4
B-1348 Louvain-la-Neuve, Belgium

1. INTRODUCTION

We briefly describe key aspects of the version base management system which is at the heart of the PMS product management system, a customizable system for

(1) managing software products having different albeit related representations at the different phases of their life-cycle,

(2) managing some aspects of the development and maintenance process of software products.

Aspect (1) entails the modeling of derivation/dependency relationships between versions of the components of a product. Aspect (2) is concerned with the control and active system support of development and maintenance operations on the software product according to the procedures and methods in use in particular organizations.

The version base (management system) can be viewed as a dedicated object management system, where the stress is put on the notion of versions, their description in terms of user-defined attributes and references to other versions for representing derivation relationships. The basic facility for controlling and supporting the evolution of a product is the attachement of user-defined policy rules to operations like version check-in/out of versions and updates of their attributes. These policy rules are basically a way to ensure that operations on the version base are according to the procedures of the organization, by specifying preconditions on the operations and actions to be triggered when these operations are invoked.

2. THE VERSION BASE MODEL

The primary kind of object in the model is the version. Versions are grouped in families. Depending on the point of view, a family is a set of versions, or it represents an "abstract" or "generic object" which can be instanciated by each of the versions in the family.

The versions in families are either atomic objects (files), or complex ones (configurations grouping atomic objects and/or configurations). Atomic and complex objects can coexist in a family, e.g. allowing for alternative versions of a component to be either a simple file or a subconfiguration.

Families live in a hierarchic name space similar to a UNIX directory tree. There is no constraint on the form of this tree, i.e. no notion of schema or structured type at this level. The hierarchy is just a means of structuring the name space for the components (families and versions) which are shared or reused by different configurations. The family hierarchy is orthogonal with the grouping of components in configurations.

Families are typed. The type includes: definition of attributes for the family, definition of attributes for the versions of the family, definition of policy rules on the operations in the components of the family. There are also provisions for type evolution, i.e. adding/removing attribute definitions and policy rules to/from types for which (instance) families already exist. The customization of a version base to the product model and procedure of a particular organization essentially amounts to the definition and modification of family types.

Since it is not possible to systematically present the formalism for defining family types, as well as all the operations on families, versions and their attributes, in the framework of a short paper, we

limit ourselves to an informal presentation based on simple-minded examples.

Example 1: definition of version attributes and precondition on assignment of values to version attributes

The following type fragments define two types of families: Requirement and Design. Each version in a family of type Requirement has a Level attribute. Each version in a family of type Design has a Status attribute, and a SatisfiesReq attribute referring to a version in a family of type Requirement. The pre-condition in the policy rule attached to the assignment of a value to the Status attribute of a Design version imposes the constraint that if the new value is "reviewed", then the corresponding Requirement version (i.e. the one referred by its SatisfiesReq attribute) must have its Level equal to "reviewed".

```
family type Requirement is
version:
      Level enum draft, to_review, reviewed;
...
end family type

family type Design is
version:
      SatisfiesReq ref version Requirement;
      Status enum draft, to_review, reviewed;
        assign policy is
        pre-condition: new_value = reviewed →
            current_version.SatisfiesReq.Level = reviewed;
        end policy
...
end family type
```

The above pre-condition can easily be refined for stating transition constraints on the successive values of the attribute Status: typically Status can only be set to "reviewed" if the current value is "to_review" etc.

Example 2: definition of post-action on assignment of values to version attributes

The assign policy in Example 1 can be extended for stating in an imperative way in a so-called post-action that when the Status attribute is set to "reviewed", the attributes ReviewedBy and ReviewedDate must automatically be set resp. to the identity of the user performing the operation and to the current date.

```
family type Design is
version:
      SatisfiesReq ref ...
      Status enum ...
      ReviewedBy string;
      ReviewedDate date;
      Status enum draft, to_review, reviewed;
        assign policy is
        pre-condition: ...
        post-action:
            if new_value = reviewed then
                  current_version.ReviewedBy := getuser();
                  current_version.ReviewedDate := getdate();
            fi
        end policy
end family type
```

Example 3: authorization in terms of pre-conditions

The authorization mechanism on the version base is similarly rule-based. In the current version of the system it is also realized in terms of pre-conditions on the operations. Relying on policy rules for the authorization mechanism makes that the authorization scheme is 2-level: the policy rules can only be updated by the system administrator; the operations on the version base are under the control of the policy rules attached to them. Only the second level can thus be customized.

A rule such as "only the creator of a family of type C_code is allowed to delete a version in such a family" can be expressed by having a "write_once" Creator attribute attached to the family, and a pre-condition on the delete operation.

```
family type C_code is
family:
        Creator string write_once;
        ...
        create policy is
          ...
          post-action: Creator := getuser();
        end policy
version:
        ...
        delete policy is
          pre-condition: current_family.Creator = getuser();
        end policy
end family type
```

3. ARCHITECTURE AND IMPLEMENTATION ASPECTS

At the lowest level, the operations on the version base are library functions to be used in the tools (user commands invoked from a standard UNIX shell, or a dedicated version base graphic browser). One natural consequence of the existence of a library layer is the stability of the tools if the technology used for the implementation of the version base evolves (e.g. replacing the current direct implementation on top of the UNIX file system by a commercial DBMS). More importantly, having the policies as rules in the version base makes that the customization of a system to the needs of a particular organization may hopefully be done in terms of policies and new attributes in family types at the version base level, whereas in a more conventional architecture, the tools or the library would have to be modified.

The first industrial version of the system is still in development at the time of writing. The system is server/client based. There is one server instance per version base. Besides sanity checks, the library functions linked with the client applications (the tools) essentially perform communications with the server. The client/server communication is based on ASCII messages exchanged via the BSD socket interface to TCP/IP.

The calls to library functions (operations) in the tools are grouped in transactions (there are calls for starting and committing a transaction), and there are no explicit locking operations in the client applications. Concurrency management is performed in the Server which is the only process physically accessing the version base on disks. Concurrency is managed with a 2-phase locking scheme. The lock tables are managed in the main memory of the server. The locking granularity is down to the level of the attributes. The policies are interpreted on the server side in the current transaction of the tool invoking the operation to which the policy is attached.

4. ELEMENTS FOR A DISCUSSION AND FUTURE WORK

The management of the concurrent accesses to the version base is done in the server; it is not in terms of the UNIX record locking facilities. We consider that record locking would introduce many constraints and complexities in the internal structures of the files representing the version base. Also, it would make deadlock detection and arbitration of deadlock resolution practically unfeasible. When compared with the use of commercial DBMS's a dedicated implementation allows for more specialized locking mechanisms (such as "intention" locks) and dedicated deadlock resolution schemes. It goes without saying that control by the implementors of the internal data structures for the families and memory management in the server is essential for the performance of the system.

The version base model sketched in this paper does not include the model for defining configurations and their structure (see [1]). This latter can be viewed as an extension of the former, albeit with integration aspects a.o. for ensuring the referential integrity of references to versions and families in configurations. For pragmatic reasons (support of existing configuration management tool sets), we considered the possibility to have different models for configurations on top of the version base. In this case, the referential integrity requirements of the particular configuration models can be supported in terms of delete policies on families and versions, provided those tools sets care to update used_by set attributes of the relevant families and versions.

The current type definition formalism for the families make that types are completeley independent one from the other. A mechanism for defining types by inheriting attributes and policies from existing ones will be elaborated.

Creation of new versions currently proceeds along conventional ckeck-in/out into/from user working directories; the new version of the object being developed with the help of tools invoked via conventional editors. In order to support control integration of software tools in the style of [2], we also plan to offer a dedicated shell or browser where tools can apply directly in the version base space for creating new versions, and where the tool applications are under the control of policy rules similar to those existing for controlling the version base operations.

References

1. Y. Bernard, M. Lacroix, P. Lavency and M. Vanhoedenaghe, Configuration Management in an Open Environment, in *Proc. First European Software Engineering Conference, Strasbourg, France, September 1987*, vol. LNCS 289, Springer-Verlag, 1988, 37-45.
2. M. Lacroix and M. Vanhoedenaghe, Tool Integration in an Open Environment, in *Proc. 2nd European Software Engineering Conf., Coventry, UK, September 1989*, vol. LNCS 387, Springer-Verlag, 1989, 311-322.

Saber-C: A Programming Environment for the C Language

Stephen Kaufer

Saber Software, Incorporated
Cambridge, MA

Saber-C is a programming environment for the C language that assists in the implementation, testing and maintenance phases of software development. The environment consists of a C parser which loads C files and performs static error checking, an interpreter which executes the intermediate code produced by the parser and performs run-time program checking, a debugger that provides extensible source language debugging, a incremental linker that combines source and object files, and several graphical browsers for code analysis and comprehension. Our goal was to develop an integrated environment for C that promotes prototyping and modular programming, provides comprehensive static and dynamic error detection, facilitates better comprehension of existing code, and automates many of the repetitive tasks associated with the development cycle.

The Saber-C environment is not designed to address all of the significant aspects of software development. Instead, it is designed to work with other tools in the development lifecycle, such as front-end design and analysis tools, and back-end configuration control and version management products. Saber-C's extensible user interface and its open architecture position the product to be used as an effective implementation and testing solution in a complete software development environment.

Position Paper for International Workshop on Unix-based Software Development Environment

Kouichi Kishida

Technical Director
Software Research Associates, Inc.
1-1-1 Hirakawa-cho, Chiyoda-ku, Tokyo 102 Japan
E-Mail: k2@sra.co.jp

SRA is one of the oldest software houses in Japan, and has been taking a leading position in Japanese software industry. We have developed and promoted a structured program design technique in early 1970s, and then during mid and late 1970s we have developed a variety of primitive CASE tools like PDLs, application generators, test coverage analyzers, etc.

So, it was a natural decision to introduce Unix into the company as a basis of in-house development environment. We made our final decision to "go Unix" in the year 1979, and our first Unix (3BSD on VAX11/780) was installed in the summer of 1980.

Our first project on Unix was testing a process control application program. We have developed a module test-bed for Fortran and a simple project support environment. The result was very successful as we expected. In the year 1981, after the success of this showcase project was reported at many conferences including USENIX-1981 at Santa Monica, our Unix staff received several hundreds of visitors from all over Japan.

Our in-house Unix environment has been enhanced year by year, and now we have several Vaxen and about 400 workstations (Sony/NEWS, SUN, etc). Of course these machines are connected by LANs and WANs. We are serving as one of the backbone sites for junet.

In the year 1981, I myself was nominated as the technical director of a government supported joint development project at JSD Corporation. The name of the project was SMEF - Software Maintenance Engineering Facility. The formal objective of the project was to develop and integrate a set of support tools for application software maintenance. But I have over-interpreted the word "maintenance" to include all phases of software development because there exist many changes in requirements during the first stage of application development. And also I decided to use Unix as the basis for developing and integrating tools. So SMEF became the first Japanese national software environment project upon Unix.

From a product-oriented view, SMEF did not made a great success. The outcoming tools of the project were not widely used because Unix was not popular yet in Japanese software industry around that time period. But from a process-oriented view, SMEF made a great contribution to the industry in educating many programmers who participated the project about the modern concept of software environment.

At the final stage of SMEF, MITI asked to me to prepare a proposal for next project. I have worked with my colleagues in SMEF and developed a plan to set up a laboratory to explore the potentials of distributed software environments consists of personal workstations connected by a local area network. After we proposed the plan, MITI enlarged it to make their SIGMA project (So we had to make another proposal for JSD's new project - FASET).

Unfortunately, after the budget for SIGMA was authorized by the government, it became clear that there was a big philosophical difference between MITI people and us (engineers in software houses). After a series of discussion, I quited from the project (the detailed explanation of the situation was described in an interview on Unix Review magazine). To prove the validity of our own idea, SRA cooperated with Sony to develop new Unix workstation which was named NEWS by me. The commercial success of NEWS in Japanese market was a big surprise for many people, but for me it was just a reconfirmation of my own belief.

During the last four years, I have organized and been managing an international R&D project to develop a common framework for future software design support environment upon network of workstations. The project SDA (Software Designer's Associates) is a unique international effort supported by a group of Japanese software houses. Some of the result of the project were reported in ICSE-10 (Singapore 1988) and ICSE-11 (Pittsburgh 1989).

Through the experience of this project, I am now deeply interested in the sociological aspects of environment construction and usage. At the workshop, I would like to discuss these matters.

An Environment for Real Time Applications with SDL as Basis, Integrations and Standardizations

Dr. Heinz Kossmann

Siemens AG
Zentralbereich Technik ZFE IS SOF 1
Otto-Hahn-Ring 6
D-8000 München 83
Fed. Rep. of Germany

Tel.: + 89 636 44207
FAX.: + 89 636 45111

A System Engineering Environment at Siemens AG

A System Engineering Environment is currently being integrated out of existing toolsets at Siemens AG that contains the following main functionalities and tool-support:

- Requirements Engineering is covered by the KLAR-Toolset with underlying AKL-method, a modified SADT method by assuming an object oriented viewpoint; additionally Sequence Charts are used in order to specify signal interchange of the system under consideration;
- Transition from Requirements to Design Engineering is facilitated by automated transformation;
- Design Engineering is supported by an SDL-Toolset, an object oriented extension is planned;
- Transition to Software Realization is achieved by target code generators for C (C + + in consideration) and the Siemens' automation technology language STEP5;
- System Integration is supported by an SDL-Testmonitor and Animation;
- Production is accomplished by a Make-File Generator.

In each step of the life-cycle, the quality assurance activities are performed as early as possible, because errors are expensive to remove if they are detected in later phases of the life-cycle. Also test data are generated in each step of the life-cycle in such a way, that later phases can be validated. The development of the system is conducted by stepwise refinement accompanied by transformations of system descriptions. It is important that links can be traced, at all stages of the development process, between user requirements and components of the developing system.

The European ESPRIT II Project ATMOSPHERE

The development of this environment is partially supported by the European ESPRIT II Project ATMOSPHERE, Ref #2565. Main partners of the project are CAP

Gemini Innovation, Bull, Philips, Siemens, SNI, and SFGL. The project duration is 3 years: 1 year definition phase; 2 years development phase. The Siemens' environment is one of four environment construction subprojects within ATMOSPHERE.

Environments in ATMOSPHERE are considered to be composed out of sets of toolsets. Toolsets in their turn again are sets of tools. The toolsets are hosted on an integration framework. Three axes for tool integration are defined: Presentation Integration that is concerned with the appearance of the environment to the user, Control Integration that is concerned with smoothing the boundaries between the various tools in terms of invocation or stimulation, and Data Integration that is concerned with minimising the effort required to enable data from one source to be used by all relevant consumers.

Integration in the ATMOSPHERE project is facilitated by the fact, that many of the existing tools are based on UNIX System V and that as standard presentation layer X-Windows and OSF/MOTIF has been accepted.

The definition phase of ATMOSPHERE has concluded that there are, in principle and praxis, complementary approaches to environment integration:

- a framework/architecture driven approach based on the use of a preselected integration framework
- a method/toolset driven approach based on more loosely coupled communication oriented mechanisms. This includes, for example, standard format definitions for data representation.

The development phase of ATMOSPHERE is organised in such a way that it starts from the current state of practice within the partner's organisations for the automated support for the system engineering process in terms of system engineering tools and methods as well as for integration approaches in order to progressively consolidate and standardise

- a common system engineering approach: process models and methods
- a set of complementary environment integration techniques.

The development phase of the ATMOSPHERE project is devided into the Toolset phase and the Environment phase in order to construct and validate Application-domain specific System Engineering Environments.

Description of Integrated Siemens' Toolsets

AKL

AKL (Aufgabenklaerung, it means task clarification) is a further development and adaptation by Siemens of the SADT method. AKL is used in the requirements phase for analysing and describing the requirements of the product to be developed. Finding the requirements, i.e. setting up the requirements catalogue for the product to be implemented, is not supported by AKL. An AKL task description ends with a description of the rough system structure, known in AKL as the product model.

All functions, tasks and components of the system to be developed can be included in an AKL description. AKL allows a structured description of a system with views onto management, its component structure, data flow and functionality, although still on an abstract, high level: thus data descriptions refer to communication paths and not to "data of type integer, real, etc.".

AKL is suited to the analysis and description of new tasks. It can also be used for maintenance, for instance for making extensions or modifications on products. For this purpose, a system must either be specified with AKL from the outset or

must at least be so far reproduced in AKL that the new components can be inserted into an environment described with AKL.

The four major steps of the AKL method are:

- embedding of the task in its environment,
- delimitation of the (hierarchically ordered) subtasks from the environment,
- requirements model to clarify the objectives from the users point of view,
- product model, rough system structure.

SDL

SDL, the CCITT Specification and Description Language, is a standard for specifying and describing telecom systems and data communication protocols. SDL of course also covers a wide range of general applications, which include real time, concurrency or parallelism. The virtues of SDL as such lies in its ability to describe and modularize the interaction and interfaces between any processes working concurrently with each other.

SDL gives a choice of two different syntactic forms to use when representing SDL descriptions; a two dimensional Graphic Representation (SDL/GR), and a textual Phrase Representation (SDL/PR). Both are equivalent from a semantic point of view. Experience shows that a graphic syntax tends to be more understandable; the result will be a more understandable documentation in all phases of the lifetime of a system. Thus SDL is used both for the specification of systems and maintenance of systems.

There is a proposal for an object oriented extension of SDL, OSDL, which has been submitted to the CCITT and which is currently under discussion. With the object oriented extension of SDL the reusability (versions and variants) and the compactness of specifications can be improved. At the same time, OSDL allows an earlier introduction of the SDL method in requirement engineering.

Editors for SDL/GR Block-Interaction and Process Diagrams as well as for Sequence Charts, and a graphic editor supporting AKL are currently being used at Siemens AG in the development of telecommunication systems and in the field of automation technology. The editors carry out local checks of the syntactical correctness. The direct predecessor symbols are checked, and when inserting a symbol the successor symbols are checked too. More complicated checks while editing are avoided for reasons of performance. These checks are made in separate tools.

The systems have an inherent complexity and therefore the process diagrams are complex. Thus in addition to checking the syntactical correctness there is a need for validation tools, such as to check liveness properties or to simulate the dynamic behaviour before implementation. Moreover it is essential to have a management for the resulting data. These requirements are the reason for the development of an integrated set of tools that support the design of complex real time software.

Object Oriented Architecture for Graphical Editors

A uniform architecture has been chosen to realise the individual editors. The common editor functions are concentrated in a common base editor. The reasons to proceed in this way are the following:

- Multiple developments are avoided and time for implementation is saved: the base editor comprises about 85% reusable software, the specific parts of the individual editors only 15% lines of code;

- The editors should be portable to different development environments that have uniform user interfaces and to different workstations. Therefore the interfaces to the operating system and to the menu handler should be at the same position in all editors;
- The user should be guided in the same way by the mentioned editors;
- The basic components can efficiently be stored in shared libraries.

The user interface is object oriented. The user edits logical objects that have a graphical representation. There can be dependencies between objects that are expressed by relations. Further the objects can have attributes, that can be manipulated by the user. The editors have an internal object oriented structure corresponding to the user interface. Message sending is realised by indirect addressing of functions by a table with pointers. An object description language has been developed to define object classes, attributes of objects, and relations between objects. Mechanisms, such as the concepts of subclasses and inheritance, are available to the tool designer. The system itself, however, is implemented in C.

Additionally, the base editor has a common control that provides a uniform user interface. The common control manages the menus, receives the user commands and calls the corresponding functions.

Experiences from Large Scale Applications

At present the KLAR- and SDL-Toolset are implemented on the SICOMP WS30 and Apollo UNIX workstation, porting to SUN (UNIX) and Siemens PC386 (OS/2) workstations is planned.

To date especially the SDL-Toolset has found great acceptance within both the field of automation technology as well as the telecommunication community. Whereas automation technology makes use of relatively small SDL-process diagrams, the diagrams in the development of the telephone exchange HICOM have hundreds or even thousands of symbols. Also, different subsets of SDL are used by the two groups.

A main condition for acceptance by the users is that the graphic editors have a reaction time in dialogue comparable to a text editor even for large diagrams. Further, the input of commands should be possible by menus as well as by keys for experts.

For running developments migration aids have to be provided in order to introduce the SDL specification technique.

Conclusions concerning UNIX

- UNIX with presentation layer X-Windows and OSF/MOTIF facilitate as international standards the cooperation of toolset vendors;
- UNIX Workstations allow the development of powerful toolsets;
- The shared library concept allows the effective use of basic components like the Siemens' common base editor for graphical languages;
- It is a disadvantage, that the X-Windows and OSF/MOTIF standard came very late;
- Standards for control integration facilities are still missing.

A Reflexive C Programming Environment

T. J. Kowalski, C. R. Seaquist, H. H. Goguen, B. Ellis, and J. J. Puttress
AT&T Bell Laboratories
Murray Hill, NJ 07974-2070

J. R. Rowland, C. A. Rath, J. M. Wilson, C. M. Castillo, and G. T. Vesonder
AT&T Bell Laboratories
Warren, NJ 07060

J. L. Schmidt
AT&T Bell Laboratories
Allentown, PA 18103-6265

We have developed an interactive C programming environment (`cens`) with integrated facilities to create, edit, browse. execute, and debug C programs. At the heart of `cens` is a C source-code interpreter, `cin`, that implements correct and complete C semantics; enables rapid prototyping; performs extensive error checks; facilitates incremental update; manages multi-file programs; and provides a programmable command language.

`Cin` is oriented toward programmers who need help testing and debugging their code as they write it. `Cin` is also useful for experienced programmers who need to debug and manufacture software in a rapid prototyping environment. It facilitates their programming by:

- Strictly enforcing type checking
- Catching errors and allowing developers to correct them ''on the fly''
- Providing facilities for incremental compilation and update
- Interfacing with various product administration tools.

`Cin` decreases time spent in the debug cycle by allowing programmers to see the effects of changes immediately. Furthermore, it narrows the gap from code generation to system test by providing an environment where a module can be tested as soon as it is manufactured. Perhaps more important for large systems, `cin`'s ability to combine source and object code seamlessly minimizes run time by limiting the *interpreted* portion to routines under development, while the bulk of the program is executed as *compiled* code. Thus, programmers can use `cin` during maintenance to learn how existing modules operate by running them in interpreted mode, while the rest of the system runs in compiled mode. Finally, `cin` supports iterative development, because new routines can be individually created, tested, and efficiently integrated with the rest of the working system.

`Cin` consists of the following:

- `cin_read`, an incremental parser and analyzer for source code
- `cin_compile`, an optimizer
- `cin_load` and `cin_unload`, an incremental loader for object and library code
- `cin_eval`, an evaluator
- `cin_print`, a universal printer.

These tightly coupled routines are the foundation of the debugger tool kit.

`Cin` command language is identical to the C language. Like the C language, `cin` is extended by predefined routines and variables. We have found that this makes it easy to customize and interface `cin` to existing software product environments. See Tables 1 and 2 for a complete list of `cin` user-modifiable routines and user-accessible variables.

Interactive programming environments are not a new idea. Environments for the LISP,[1,2] PASCAL,[3-5] PL/C,[6,7] and Smalltalk[8,9] languages have existed for several years. These environments are designed to improve the productivity of programmers and the quality of their programs.[1,4,10] Within the last few years,

Table 1. CIN USER-MODIFIABLE ROUTINES

`cin_break`, `cin_error_code_set`, `cin_pop`, `cin_return`, `cin_stopin`, `cin_system`, `cin_unbreak`, `cin_unstopin`	handle breakpoints
`cin_run`, `cin_step`, `cin_stepin`, `cin_stepout`	step through code
`cin_make`	maintain collections of files
`cin_spy`, `cin_unspy`	watch variables for access or modification
`cin_unwrapper`, `cin_wrapper`	watch routines for calls and return
`cin_epp`, `cin_eprint`, `cin_pp`, `cin_slashify`	print routines
`cin_dump`	save the environment as an executable program
`cin_log`	record user sessions
`cin_find_ident`, `cin_find_nlist`, `cin_call_graph`, `cin_info`, `cin_info_set`, `cin_ltof`, `cin_sync`, `cin_trace`	provide information
`cin_view`	control the symbol tables of source code
`cin_quit`	exit
`cin_reset`	start a program from its initial state

Table 2. CIN USER-ACCESSIBLE VARIABLES

`cin_argc`	number of arguments
`cin_argv`	arguments
`cin_err_fd`, `cin_in_fd`, `cin_out_fd`	`cin`'s file descriptors
`cin_filename`	name of the currently executing file
`cin_level`	depth of recursive interpretation
`cin_libpath`	search path for libraries
`cin_lineno`	line number in the currently executing file
`cin_prompt`	user prompt
`cin_stack`	run time stack
`cin_views`	list of loaded source and object files

work has started on various pieces of a C program environment, including syntax-directed editors,[11] smart compilation, interpreters,[12, 10] debuggers,[13, 14] and browsers.

Our solution combines a multi-window editor and browser, an on-line advisor, a C source-code interpreter, and an incremental object file loader. Our programming environment has an open architecture; that is, the pieces work separately as well as together. This means that any one utility—advisor, browser, editor, interpreter, or loader—can be used by itself, or with one or more additional tools in a customized environment. Care has been taken to keep the core of `cin` uncluttered by placing much of the functionality in libraries, which can be tailored and customized by developers. For example, both the single step and trace packages are libraries. Thus, we have created an open architecture for rapid prototyping of a custom development environment as well as the application themselves. Cens supports rapid prototyping because existing modules can be found and modified easily; the effects of changes can be seen and tested immediately; specifications can be created and tested through prototypes; and separate tools all work together as a single unit.

USING CENS

Once loaded into `cin`, the program text and data are available for interpretation, analysis, and modification. Combined with the open architecture, this reflexive property, which allows any program to analyze and modify itself, facilitates the creation of environments similar to those available in LISP. These environments can be tailored into platforms for conducting research in software engineering. Using `cin`, we have explored the creation of both static and dynamic analysis tools. Some of these tools examine code organization and redundancy by clustering functions according to the types of manipulated variables. Other tools have been written that record the range of values for designated variables to study automatic synthesis of invariants and test generation.

The tools in the `cens` interactive C programming environment have been mixed and matched to form several customized programming environments throughout AT&T.[15]

- Interactive Graphics—`Cin` is used to develop the Xt-based OPEN LOOK® "widget" set. `Cin` provides incremental compiling and loading for developing and testing Xt widgets. It has reduced the typical load and test time from 3 minutes to 30 seconds. It also eliminates the need for disk storage of large numbers of executable unit tests. Similarly, `cin` is used for MIT X Windows™ widget synthesis and editing.
- Large Multiprocess Systems—`Cin` is used to debug and test the switched access remote test system multiuser environment. When combined with `sable` and `nmake`, `cin` provides a software manufacturing environment where errors are caught and corrected "on the fly" without reloading the multiuser environment. `Cin` has replaced the five- to ten-minute reload of the multiuser environment with an incremental update of under a minute. A similar effort is going on in the AT&T Definity® 75 development team.
- Unit Testing—`Cin` is used to build a unit test environment for the AT&T AUTOPLEX® cellular telecommunications system. It provides a simulation of the diskless standalone environment needed to test and integrate software modules. Regression, coverage, error-path, low-level integration, and unit tests are performed, and unit test histories are recorded.
- Incremental Loading And Compiling—`Cin`'s incremental loader is used to add customized functions at run time to AT&T's schematic capture system, `schema`; a simulation system, `midas`; and AT&T's place and route tool, `ltx2`. `Schema` uses `cin`'s incremental loader to decrease execution time by a factor of more than 27 (i.e., 2700 percent) and memory usage by a factor of 260 percent. The interpreter is under consideration as a simulation engine for behavioral synthesis work.
- Integrated Debugging Environments—`Cin` is used as a debugger for the software development assistant environment. Combined with the object generation system, MIT X Windows, and the `Andrew` system, it provides a software manufacturing environment for unit testing in the Definity 75 system.
- Program Generation Environments—`cin`'s incremental loader is used to add synthesized functions at run time in the BriefCase project. Combined with an object-oriented database, the loader provides software environments where object-oriented database properties implemented in the C language can be modified and then reused dynamically.

BUILDING CENS

The portability provided by the UNIX® system has been crucial to the success of `cin` and a source of frustration at the same time. Currently, `cin` is deployed on variety of UNIX platforms (see Table 3). The large number of supported platforms confirms the ease with which applications can be ported between UNIX systems. Some customization for each platform is, however, necessary. This effort requires 20 to 100 lines of assembly code for stack frame manipulation along with the handling of `setjmp` and `longjmp`. In addition, because the Common Object File Format (COFF) and the research `a.out` format are not as common across UNIX systems as one might believe, the incremental loader may also require modification.

Table 3. SUPPORTED PLATFORMS

Amhdal 580	SVR2.0.1	Sun 386	SunOS4.0.1
PC 6386 WGS	SVR3.2	Sun 3	SunOS4.0.3
7300	SVR3.51	Mips	UMIPS4.0
Sparc Station 1	SunOS4.1	Sun 4	SunOS4.0.3
3B2	SVR3.1	3B15	SVR3.1
Vax	BSD4.3	Vax	10th Edition
Vax	SVR3.1		

One of our central goals—seamless compatibility between compiled and interpreted code on each platform—creates another compatibility problem. When `cin` is built for a particular platform, the idiosyncrasies and bugs of the C language compiler are introduced. This allows developers to move easily between compiled and interpreted code.

Two issues relating to the UNIX system have limited the general power of `cin`. An early version of `cin` allowed a program to be run backwards as well as forwards. This is similar to the debugging environment provided by many rule-based systems. Unfortunately, the lack of a journaling capability in the UNIX system makes the reversing of many system calls difficult, which in turn severely limits the usefulness of running backwards. A second issue arises from the extensive use of pointers within shared-memory regions by large real-time applications. This diminishes the accuracy of the extensive pointer analysis provided by `cin`.

FUTURE CENS

We are currently using `cens` to explore interface design, pedagogical methods, and programming environment technology. Instead of creating an application-oriented language, we provide an interpretive C language interface that allows programmers to use C code to enter, manipulate, and debug an application. This design choice both decreases training time for the application and increases reuse of existing interface code. `Cens`, used with `bonsai`, a powerful graph browser and animator system, helps students learn the C language by quickly pinpointing programming errors; providing data structure animation facilities; and reducing the number of languages they must master. (Because the command language of `cin` is C, the burden of teaching yet another debugger language to the novice programmer disappears). This easy-to-use instructional environment helps students concentrate on programming concepts and algorithms rather than language syntax. Finally, our current research in programming environment design explores the use of distributed interpretation. For example, a distributed version of `cin` could reduce the debugging and maintenance cycle for switch-development applications like the Definity 75 system. We are also interested in developing a programmer's toolbox. Currently, we are enhancing `cens` with additional facilities for large multiprocess systems, e.g., intelligent testing, reverse engineering, and graphical interfaces.

REFERENCES

[1] Teitelman, W. and Masinter, L., "The INTERLISP programming environment," *Computer* **14**(4), pp. 25-33 (April, 1981).

[2] Sandewall, E., "Programming in an Interactive Environment: The "LISP" Experience," *Computing Surveys* **10**(1), pp. 35-71 (March, 1978).

[3] Delisle, N. M., Menicosy, D. E., and Schwartz, M. D., "Viewing a Programming Environment as a Single Tool," *Software Engineering Symposium on Practical Software Development Environments*, pp. 49-56 (April 24, 1984).

[4] Reiss, S. P., "Graphical Program Development with PECAN Program Development Systems," *ACM SIGPLAN Notice* **19**(5), pp. 30-41 (May, 1984).

[5] Brown, M. H. and Sedgewick, R., "A System for Algorithm Animation," *Computer Graphics* **18**(3), pp. 177-186 (July, 1984).

[6] Archer, J. Jr. and Conway, R., "COPE: A Cooperative Programming Environment," TR 81-459, Cornell University, Department of Computer Science (June, 1981).

[7] Teitelbaum, T. and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM* **24**(9), pp. 563-573 (September, 1981).

[8] Goldberg, A. J. and Robson, D., *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, Reading, MA (1985).

[9] Tesler, L., "The Smalltalk Environment," *BYTE*, pp. 90-147 (August, 1981).

[10] Feuer, A. R., "si - An Interpreter for the C Language," *USENIX Summer Conference Proceedings*, pp. 47-55, USENIX Association, Portland (June, 1985).

[11] Horgan, J. R. and Moore, D. J., "Techniques for Improving Language-Based Editors," *ACM SIGPLAN Notices* **19**(5) (May, 1984).

[12] Kaufer, S., Lopez, R., and Pratap, S., "Saber-C: An Interpreter-based Programming Environment for the C Language," *USENIX Summer Conference Proceedings*, pp. 161-171, USENIX Association, San Francisco, CA (1988).

[13] Adams, E. and Muchnick, S. S., "Dbxtool A Window-Based Symbolic Debugger for Sun Workstations," *USENIX Summer Conference Proceedings*, pp. 213-227, USENIX Association, Portland (June, 1985).

[14] Cargill, T. A., "The Feel of Pi," *USENIX Winter Conference Proceedings*, pp. 62-71, USENIX Association, Denver (January, 1986).

[15] Belanger, D. G. and Wish, M. (Eds.), *AT&T Technical Journal: Software Productivity* **69**(2), AT&T, New York, NY (March/April, 1990).

Integrated Test Management

David Lubkin
Apollo Systems Division
Hewlett-Packard
300 Apollo Drive
Chelmsford MA 01824

Abstract

Efforts to improve software quality and standardization have led the UNIX community to growing use of software validation and test suites. The issue is then how to manage test activities so that they don't introduce more problems than they solve, due to the time required to run the tests, and uncertainty over whether all necessary tests were run against the right pieces of software.

These turn out to be much the same issues that led to version control and configuration management for source code. This paper discusses the application of an established UNIX software configuration management system, the Domain Software Engineering Environment (DSEE),[4,9-12] to issues of test management.

Introduction

As the UNIX crowd matures, we're gradually discovering that hacking won't, er, hack it any more. Concerns over software quality and standardization have led to widespread use of validation suites and test suites.

This is good. On the other hand, it's a mixed blessing. How can you remain productive when it takes 100 hours to run a test suite? What happens when a computer that's running the tests crashes? How can you test multiple configurations? How do you ensure that tests are run when they're supposed to be run? How do you know that you've tested the right software?

DSEE

Previous writers have alluded to some of these issues, but none have provided any answers. [1-3,7,8,13,14,16,17]

Like everyone else, we in the DSEE group were faced with these problems. Our solution was to use our own product to control its test process.

The Domain Software Engineering Environment (DSEE) is one of the most popular tools for UNIX source control and configuration management. It is in use at over 6000 sites world-wide, and throughout Apollo Computer. We estimate that it is managing over one billion lines of code in all.

DSEE is designed to deal with the particular problems of large-scale software development. Some of these problems are: working on multiple development efforts in parallel, maintaining existing releases while working on new code, coordinating code and documentation, reducing the time it takes to build a program, and knowing precisely and forever which tools and versions of source code were used to create a binary.

DSEE unifies development and maintenance for large-scale systems (the largest customer so far has 30,000,000 lines under DSEE control). It supports project development in a distributed computing environment, and works with any programming language or tool. For test management, it can be used for test script CM, as a test execution facility, and as a librarian for test results. DSEE is highly reliable, thanks to transactional file and database operations, store-and-forward message passing, journaling, and time-proven fault recovery procedures.

Source control

Source code for generating tests, test scripts, and test results can be kept as **elements*** in DSEE **libraries** (equivalent to files in a directory).

Libraries are managed by DSEE's **history manager**, which stores, controls and tracks source code evolution. It enforces concurrency control and security, and keeps a complete chronological audit trail of library modifications.

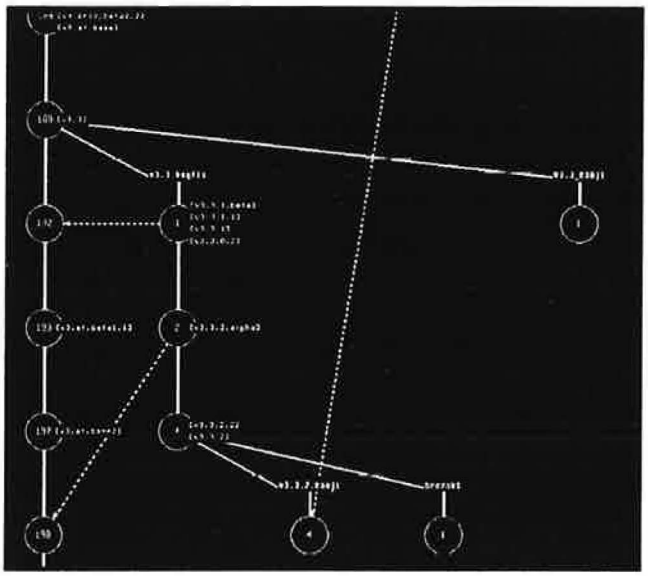
DSEE views the evolution of an element as a directed acyclic graph. Any number of people may work on the same element concurrently by making new **branches**. Their work may be kept separate or merged together with a sophisticated three-way merge command. The trunk is

* DSEE terminology is printed in boldface when first introduced.

also considered to be a branch; it is usually referred to as the **main line of descent.**

Branches have names, like v3.3.bugfix or BuzzWord. **Versions**, or generations, on a branch have version numbers, but can also be given one or more names.

The evolution of an element can get pretty complicated. That's why DSEE has a command to graphically display the structure of an element's evolution, including branch and merge points. The show derivation command also has options to 'prune' the display of the tree.



Versions are stored together efficiently. DSEE's interleaved forward deltas and compression let 50 to 200 versions of source code be stored in the same space as two ordinary text copies. Binary files can be stored under DSEE, but they are not stored as compactly as text files. Any version of either can be read in a single pass through a DSEE history file.

Transparent access to versions

Any DSEE element can be read directly, without "checking it out" from the library, thanks to Domain/OS's typed file system,[5,15] either from an Apollo or across a transparent remote file system like NFS, DECnet, or Domain/PCI. This lets ordinary applications like cc and troff operate on objects under DSEE control without modification.

Under Domain/OS, each file has a type. DSEE elements are of type case_hm. Each type has an associated manager that performs all I/O for that type.

When asked to read a pathname that ends with a DSEE element's name, the case_hm type manager returns the most recent version on the main line of descent. If you want to read some other arbitrary version, you can either explicitly name it or have it implicitly associated with the element.

If the pathname continues beyond a DSEE element, the case_hm manager interprets the remainder of the pathname as a sequence of branches and generations.

Users can also implicitly refer to a collection of versions, like those used to build some binary, or the version named motorola of each of their files. On Domain/OS, you can create a shell that is bound to that collection. Whenever you read files from that shell, the case_hm manager will access the particular versions you specified.

This mechanism for implicit reference to versions is an integral part of DSEE configuration management, described below. It's used when building and when debugging.

System models

A system model is a set of files that describes the static structure of a program or system. As such, it is similar in concept to a makefile.[6] It is written in a Pascal-like block-structured language with preprocessor constructs. System model components may either be DSEE elements or 'external' non-DSEE files.

For each component of the system, the model expresses:

- Source dependencies, i.e., what it 'includes'.
- Translation rule, i.e, how to compile it. Source files and binaries are referred to through macros that DSEE expands at build time.

  There are no restrictions on the contents of a translate rule. Rules could be used for interactive testing, UI testing, embedded systems testing, and the testing of distributed or heterogeneous applications.
- All possible translation options, e.g., -debug, -opt.
- Tool dependencies, i.e., tools required for the translation.
- Subcomponents that must be built first.
- Where to put the results of translations.

A system model for test execution reflects the internal dependencies between tests. Any necessary set-up can be placed in a component that is depended on by all other components. Any necessary clean-up can be placed in a component that depends on the tests.

A system model for testing can be structured to reflect more than one characteristic of each test, e.g., function, author, hardware requirements, etc. Users can then select tests to run based on any combination of those characteristics, e.g., "run all parsing tests, all tests written by Joe, and all tests for the chmod command."

The system model compiler supports a number of preprocessor options, including preprocessor variables that can be used to configure the tests for multiple targets or configurations. At Apollo, the system model for the Domain/OS operating system supports over a dozen different hardware variants

Incorporating testing into development

Tests that must be compiled before they are run can also be described by system models. They can either be in the same system model as runnable tests or in separate models. Either way, they will only be rebuilt when they change.

For that matter, it is reasonable to structure one's system models so that whenever you build a program you're working on, the test suites are automatically run. Since you don't want to do this while you're still debugging, you can tie the test suite execution to a model variable that is only enabled when you're ready to test.

Threads

Configuration threads describe which versions and translator options to use when building each component in the system model. They are written in a powerful rule-based language.

By changing their thread, users can quickly switch among development projects, fixing bugs in old releases, and making special versions for particular targets or customers.

Threads can be used to select versions of tests to run, and what values to use for test parameters or configuration.

Configuration thread rules may be:

explicit	use version 5
dynamic	use the most recent version
referential	use the same version used in rev2
contextual	use x.h[7] for P, otherwise x.h[6]

There is also a **model thread**, which applies to system models. Models are usually stored within a DSEE library so that you have a precise description of the structure of a program at any time in its history. They are often composed of several files, or **model fragments**. The model thread describes which versions of model fragments and conditional compilation options (**targets**) to use when interpreting the system model.

Building

DSEE combines system models and configuration threads to make a **bound configuration thread** (BCT), which is a permanent record of the versions and options used to build the component. BCTs are kept along with build results (test results, in our case) in special directories called **pools**. When users type build, DSEE looks in their pools for BCTs that match the components they've asked to build. DSEE keeps a number of earlier builds in the pool and attempts to reuse them. If it can't, and the pool is full, the least recently used build of that component is purged out to make room for a new one.

This ensures that tests that have already passed do not need to be rerun. There is a record of tests run, exactly which tools were used, and exactly which versions were used.

DSEE sequences the required tests according to their inter-dependencies, and starts running them. As your translation script executes, it reads the desired version of each DSEE element directly out of its library.

After a program has passed all tests, the results can be **released**, creating a permanent record, linking tests with the versions of software being tested, and from there, to the source code that makes up the software.

Running tests in parallel

DSEE has the ability to invoke translations of components on up to 50 Apollo computers at a time. DSEE tracks the CPU utilization of candidate build computers, and starts builds on the least loaded machines.

This parallel build capability can be used to run tests in parallel. If the tests are run in a window, DSEE continually displays a summary of how many tests remain to be run, how many succeeded, how many failed, and which ones are currently in progress.

The performance improvement provided by parallel building can be quite dramatic. For example, the Ada Validation Test Suite of 2693 tests used to take 80 hours to run. With parallel building, it now takes 17 hours. Other applications have shown a speed-up of 35X.

One complication introduced by parallel building is that there is no guarantee that all of the computers used to compile a program have the same versions of tools and system files.

If your test program were compiled with an arbitrary collection of stdio.h's, you might spend weeks tracking down what was wrong. DSEE avoids this problem by interpreting all /-relative pathnames in translate scripts relative to a **reference directory.**

Test interpretation

There are a variety of commercial and home-grown tools for test analysis. These tools can either be run immediately after test execution by invoking them in a system model as a post-processing step, or run later on, after retrieving the test results from their pool or release area.

To help the latter process, DSEE has two programmatic interfaces apart from its interactive UI. /lib/dseelib can be dynamically bound into a program, which can then issue DSEE commands:

```
dsee_$set_library("/foo/bar", 8, [], st);
if st.all = dsee_$path_component_too_long then
```

This callable interface was used to write dsee_server_c, an example shipped with DSEE. A client program, ds, sends IPC messages containing DSEE commands to the server, which executes them through the procedure dsee_$cmd. The net result is that shell control constructs can be used to postprocess the results of DSEE commands.

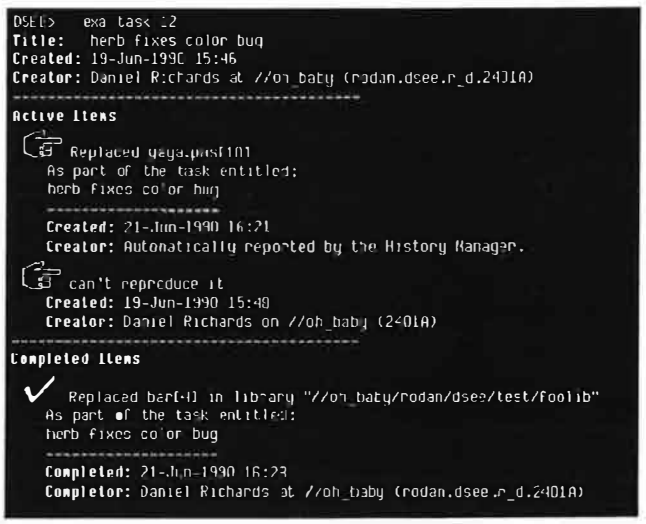
```
# ds set library /foo/bar -n
# if [ "$?" -eq DSEE_PATH_COMPONENT_TOO_LONG ] then
```

The two are often used to customize DSEE to personal and corporate methodologies. dsee_server_c is especially popular with UNIX shell wizards.

Tasks

The DSEE **task manager** plans and tracks the low-level steps taken toward achieving a high-level goal, like "fix bug #1332 for Lockheed." It automatically records DSEE actions made on behalf of the goal, including all source modifications toward that task, your comments at every step, and a complete record of all tests run to verify the bugfix. It can also be used to record non-DSEE tasks to do or already done, like "test the new user interface with a focus group." For tracking recurring activities, new tasks can be created using a task template facility.

In the short term, **tasks** can be used by team members and managers to track progress toward goals. Later on, they provide a historical record that can be useful for resolving similar problems, and to help new team members quickly learn what has been done and what steps were taken.



Monitors

The **monitor manager** tracks people-oriented dependencies. The user describes what DSEE elements to monitor, who can or can't activate the monitor, and what should happen when the monitor is activated. The monitor then remains dormant until it is triggered by changes to the elements being monitored. At that time, the monitor can:

- send a mail message to one or more users
- add a new task to some users' task lists
- send an alarm window to anyone on the network
- execute an arbitrary shell script

Monitors can be used alone or with tasks to track the progress of the test development and execution cycles.

Current deficiencies

This solution is not with flaws:

- There is no flexibility in the criteria by which candidate build computers are selected.
- It is cumbersome to use DSEE to test software targetted for non-Apollo platforms.
- DSEE is not available on non-Apollo platforms.
- There is no way to detect that a test is in an infinite loop.
- DSEE will not automatically rerun tests that fail due to hardware or network trouble.

References

[1] Beizer, Boris, *Software System Testing and Quality Assurance*, NY: Van Nostrand Reinhold, 1984.

[2] Beizer, Boris, *Software Testing Techniques, Second Edition*, NY: Van Nostrand Reinhold, 1990.

[3] Bryan, William L. and Siegel, Stanley G., "Configuration Management of Software Testing," 1984 Software Maintenance Workshop, IEEE, 1984.

[4] Chase, Robert P., Jr. et al, Device for Managing Software Configurations in Parallel in a Network, U.S. Patent No. 4,951,192.

[5] Leach, Paul J. et al, "The Architecture of an Integrated Local Network," *IEEE Journal on Selected Areas in Communications*, Nov. 1983.

[6] Feldman, S. I., "Make — A Program for Maintaining Computer Programs," *Software Practice and Experience*, Apr. 1979.

[7] Hollowich, Michael, and Borasz, Frank, "The Software Design and Verification System (SDVS): an Integrated Set of Software Development & Management Tools," *Proceedings of the IEEE 1976 National Aerospace and Electronics Conference*, May 1976, Dayton, Ohio.

[8] Hornbach, Katherine, "Software Tools: A Way to Control Complexity on Large Software Projects," *Proceedings of the IEEE 1985 National Aerospace and Electronics Conference*," May 1985, Dayton, Ohio.

[9] Leblang, David B. et al., "The DOMAIN Software Engineering Environment for Large-Scale Software Development Efforts," *Proceedings of the First International Conference on Computer Workstations*, Nov. 1985, San Jose, Calif. IEEE Computer Society.

[10] Leblang, David B. and Chase, Robert P., Jr., "Parallel Software Configuration Management in a Network Environment," *IEEE Software*, Nov. 1987.

[11] Leblang, David B. et al, Computer Device for Aiding in the Development of Software System, U.S. Patent No. 4,809,170.

[12] Lubkin, David, "DSEE: The Cure for CM Blues," *Proceedings of the First Hewlett-Packard Software Engineering Productivity Conference*, Aug. 1990, San Jose, Calif.

[13] Pirie, J.W., "Keep cost and confusion low (configuration management)," *ComputerWeekly* No. 948, Jan. 31, 1985, page 33.

[14] Pyper, Walter R., "Historical Files for Software Quality Assurance," *Proceedings of the Computer Performance Evaluation Users Group (CPEUG) 17th Meeting*), Nov. 1981, Washington, DC.

[15] Rees, Jim, et al, "An Extensible I/O System," 1986 Summer USENIX Conference.

[16] Thoreson, Sharilyn A., "The automated software development project at McDonnell Aircraft Company (the Software Factory)," *Proceedings of the IEEE 1989 National Aerospace and Electronics Conference*, May 1989, Dayton, Ohio.

[17] Tobias, R. W. et al, "Configuration Management Techniques for Automatic Testing," AUTOTESTCON 80.

A Text-Based Reuse System for UNIX Environments

Yoëlle S. Maarek
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
yoelle@ibm.com

Mark T. Kennedy
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
mtk@ibm.com

1 Reuse and Software Development Environments

Software reuse is widely believed to be a promising means for improving software productivity and reliability [6], however it is not practiced enough. One of the reasons for the only moderate success of reuse is the lack of tools promoting it. Help/reuse mechanisms should be available in every elaborate software development environment so as to permit locating and understanding software components.

The two basic requirements for achieving reuse are: (1) to provide a sufficient number of components over a spectrum of domains that can be reused as is (*black-box* reuse) or easily adapted (*white-box* reuse), and (2) to organize components such that existing code closest to the users' needs is easy to locate.

It is widely recognized that the UNIX toolkit is one of the most successful examples of reuse as far as the first requirement is concerned. Tools were written and documented in order to promote black box reuse and the first reuse requirement has clearly been fulfilled. However, as far as the second requirement is concerned only very primitive search and retrieval means are provided with UNIX: `man`, `man -k` or `apropos` and `grep`. Elaborate intelligent systems such as UC, [15], or hypertext tools such as INFOEXPLORER, a CD-Rom Hypertext Information Base Library, which comes with AIX (IBM's UNIX version) in the IBM RISC System/6000 series, have been provided in some UNIX environments. Unfortunately such tools are very expensive to build, require a great deal of manual encoding, and are not easily extensible. As a consequence, they are not available on all UNIX systems.

2 Previous Work

Previous efforts for building reuse systems can be roughly classified into two approaches: the information retrieval (IR) approach and the artificial intelligence (AI) approach.

There has been much work in IR geared towards analyzing natural-language text, and a large variety of techniques have been described for indexing, classifying and retrieving documents [12]. IR-based reuse tools draw information only from the structure of the documents. No semantic knowledge is used and no interpretation of the document is given: the reuse tool attempts to characterize the document rather than understand it. There has been relatively little effort in this direction: [7],[2], [3]. In contrast, examples of AI or knowledge-based reuse tools are numerous: [11], [16], [1], [5], [13], etc. In this approach, the reuse tool aims at understanding the queries and the documents before providing an answer. AI-based systems are often "smarter" than IR systems: some of them are context sensitive and can generate answers adapted to the user's expertise. As a tradeoff, they require some domain analysis and a great deal of pre-encoded, manually provided semantic information. The AI approach can be useful in some applications for which the system's intelligence is crucial. However, in our context of software libraries, we prefer the IR approach for obvious reasons of cost, portability and scalability.

3 An IR tool for reusing UNIX tools based on the manual pages

In this paper, we claim that simple reuse tools can be automatically built for any UNIX environment without requiring expensive pre-encoded knowledge or domain analysis, simply by using an IR approach. We describe a technology for automatically building such reuse tools, and present a system that embodies this approach while surpassing the performance of more expensive tools. We propose to take advantage of a rich source of information that exists in most UNIX environments: the manual pages. Manual pages provide a great deal of conceptual information about the functionality of tools. However, this information is contained only implicitly and is not usable as such. In order to extract information from manual pages, we use a refined version of an indexing scheme introduced in [10]. The atomic indexing unit we use is the *lexical affinity* (LA). An LA between two units of language stands for a correlation of their common appearance [4]. We build LA-based signatures for each manual page in three steps.

In a first step, LAs are automatically extracted from the manual page by using a co-occurrence compiler. LAs are then stored, together with their frequency of appearance, in their canonical form (each word[1] is represented by its inflectional root). Table 2 presents the set of LAs extracted from the manual page of **mv** in AIX. For the sake of comparison, a list of the content words also ranked by frequency is shown in Table 1. The LAs clearly outperform single words in terms of meaningfulness. However, LAs still suffer from noise due mainly to words frequently occurring in the domain.

In a second step, we compute the ***resolving power*** of LAs, which is defined as , $\rho_d(w, w') = f_d \times \text{INFO}(w, w')$ where f_d is the frequency of the LA (w, w') in document d, and INFO is the quantity of information of the two words in the entire domain (the set of manual pages). Taking into account the quantity of information of words, as defined in information theory, permits us to reduce the influence of context words.

Finally, in order to compare the resolving power of LAs across manual pages, and keep only the most representative LAs, we transform ρ values into their z-score $\rho_z = (\rho - \overline{\rho})/\sigma$, where $\overline{\rho}$ is the average ρ in the document, and σ is the standard deviation, and we keep in the signature the LAs with a ρ value bigger than the average plus the standard deviation (peak in the distribution of ρ).

content words	freq
file	30
directory	14
mv	11
files	8
new	7
name	7
move	7
newname	6
is	6
system	5
...	...

Table 1

LAs	freq
file move	9
be file	8
directory file	7
file system	5
file overwrite	5
file mv	5
file name	4
name path	3
do file	3
directory move	3
...	...

Table 2

LAs	ρ_z
file move	8.38
file mv	4.36
directory file	4.03
file overwrite	3.87
directory move	1.98
file system	1.95
mv rename	1.71
move mv	1.58
different file	1.40
name path	1.33

Table 3

Table 3 ranks the LAs by ρ_z-value. As we see, the most important concept *(file move)* has a resolving power clearly greater than the following LAs. Moreover, some noisy LAs such as *(do file)* or *(be file)* have disappeared because *both* words involved in the LA have a low quantity of information.

LA-based signatures are thus automatically built for each manual page. All associated software components can then be classified, stored, compared, and retrieved according to these indices. They can even be organized into hierarchies for browsing, using clustering techniques [9]. At the retrieval stage, the user

[1] We only consider LAs involving *open-class words*, or *content words* (nouns, verbs, adjectives and adverbs) as meaning-bearing, whereas LAs involving *closed-class* words (pronouns, prepositions, conjunctions and interjections) are not.

can specify a query in free-style natural language, that query being indexed by using the same indexing technique. A set of LAs is extracted from the query and directs the repository search. Various ranking measures can then be used in order to select the best candidate for a particular query [14].

4 Repository structure

We use a trie data structure to store an inverted index of the set of LAs. Due to the nature of our measure of similarity between queries and components in the repository, we need to store not only LAs but also single words occurring in signatures. LAs as well as single words are considered as strings of characters and are stored as a path to a cell containing a list of pairs (p, ρ) where p is a pointer to the document whose signature contains the index, and ρ is its resolving power in the considered document.

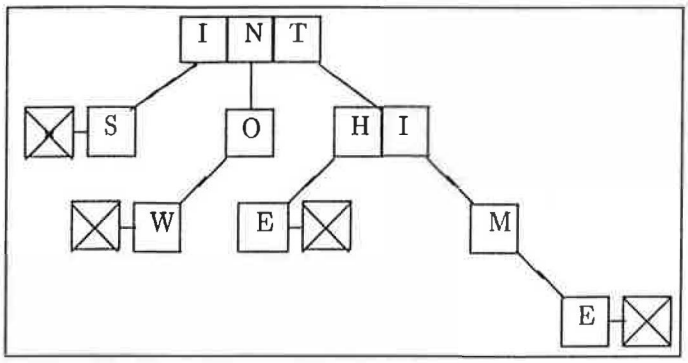


Figure 1: Example of trie storing the words "now", "is", "the", "time"

An example trie data structure is shown in Figure 1 where the words "now", "is", "the", and "time" are stored. The crossed boxes indicate a cell that stores the list of (p, ρ). Using a trie data structure is advantageous in our case because of the numerous repeated prefixes. We also optimized the data structure in the two following ways:

- Every node in the trie consists of a list of pointers labeled by a single letter. We have performed a statistical analysis of the distribution of letters in the corpus of signatures and ordered the labeled pointers accordingly[2] (from highest to lowest probability of occurrence) .
- We linearize the trie via a breadth first walk. This clusters the most frequently examined nodes together (they correspond to the most shared prefixes) thus amortizing the cost of faulting into memory.

Another possible optimization would be to eliminate leaf chains. An analysis of two corpora showed that the space savings would be significant. This follows the intuition that suffixes are limited in number in English. However, this last optimization makes the update operations more complicated and performance was already adequate, even on an RT.

5 Implementation

Using this methodology, we have built a prototype reuse/help system for AIX that accepts free-style natural-language queries. This system consists of a collection of shell scripts and programs that produce

[2] Figure 1 shows the labeled pointers in lexicographic order only for example purposes.

the index and other information necessary to perform queries. It depends on the availability of an efficient file-mapping facility such as that provided by AIX, SUN, Mach, and presumably 4.4BSD. We treat an index like a persistent data structure in the programs that build and query them. We are currently: 1) integrating this indexing facility into RPDE, a structured software development environment at the IBM T.J. Watson Research Center and 2) re-doing the original implementation to be simpler (one program) and to have an X interface (Motif-based) as well as a command line interface.

6 Evaluation

In order to evaluate performance, we have considered three criteria: (1) efficiency, (2) maintenance effort and (3) retrieval effectiveness.

1. In our RT-based system, an index built using 18,000 LAs describing 1,100 AIX on-line manual pages consumed 2.5 Megabytes of disk space. Test queries involving 5-15 LAs took approximately 2.5 seconds. Profiling the execution of the query program showed that the time to perform the query was dominated by the time to map the repository file into the address space of the query program. The lookup operations and the printing of the LA-file name pairs consumed almost no time in comparison. On the IBM RISC System/6000, the same queries took 0.15 seconds (the 6000's implementation of file mapping is "lazier" and hence more efficient). Let us note also, that our system is, independently of implementation considerations, much faster than tools like INFOEXPLORER (see below) since it does not explore the whole documentation database, but only a smaller signature repository.

2. The indexing is performed entirely automatically and the insertion of new components can be done incrementally. Kaplan and Maarek in [8], have proposed several algorithms that allow to incrementally update a repository of LA-based indices when inserting, deleting or modifying components.

3. We have compared the performance of our system to INFOEXPLORER, a hypertext-based library system for AIX on the RISC System/6000 series. INFOEXPLORER provides not only hypertext links between pieces of the AIX documentation library, but also search and retrieval facilities using word-based IR techniques. We conducted a comparative test between both systems by evaluating their recall and precision rates. We expected INFOEXPLORER to have a very high recall but low precision since it explores the entire documentation database. We determined that, thank to its indexing scheme, our system achieves a recall rate as high as INFOEXPLORER and a precision rate 15% better than INFOEXPLORER while being much less expensive to build.

References

[1] B.P. Allen and S.D. Lee. A knowledge-based environment for the development of software parts composition systems. In *Proceedings of the 11th ICSE*, pages 104-112, Pittsburgh, PA, May 1989.

[2] S.P. Arnold and S.L. Stepoway. The reuse system: cataloging and retrieval of reusable software. In W. Tracz, editor, *Software Reuse: Emerging Technology*, pages 138–141, Computer Society Press, 1987.

[3] B.A. Burton, R. Wienk Aragon, S.A. Bailey, K.D. Koelher, and L.A. Mayes. The reusable software library. In W. Tracz, editor, *Software Reuse: Emerging Technology*, pages 129–137, Computer Society Press, 1987.

[4] F. de Saussure. *Cours de Linguistique Générale, Quatrième Edition*. Librairie Payot, Paris, France, 1949.

[5] P. Devanbu, P.G. Selfridge, B.W. Ballard, and R.J. Brachman. A knowledge-based software information system. In *Proceedings of IJCAI'89*, pages 110–115, Detroit, MI, August 1989.

[6] W.B. Frakes and P.B. Gandel. Classification, storage and retrieval of reusable components. In N.J. Belkin and C.J. van Rijsbergen, editors, *Proceedings of SIGIR'89*, pages 251–254, ACM Press, Cambridge, MA, June 1989.

[7] W.B. Frakes and B.A. Nejmeh. Software reuse through information retrieval. In *Proceedings of the 20th Annual HICSS*, pages 530–535, Kona, HI, January 1987.

[8] S.M. Kaplan and Y.S. Maarek. Incremental maintenance of semantic links in dynamically changing hypertext systems. *Interacting with Computers*, December 1990. To appear.

[9] Y.S. Maarek. An incremental conceptual clustering algorithm with input-ordering bias correction. In M.C. Golumbic, editor, *Advances in Artificial Intelligence, Natural Language and Knowledge Base Systems*, Springer Verlag, 1990. To appear.

[10] Y.S. Maarek and F.A. Smadja. Full text indexing based on lexical relations. an application: software libraries. In N.J. Belkin and C.J. van Rijsbergen, editors, *Proceedings of SIGIR'89*, pages 198–206, ACM Press, Cambridge, MA, June 1989.

[11] R. Prieto Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, January 1987.

[12] G. Salton. *Automatic text processing, the transformation, analysis and retrieval of information by computer*. Addison-Wesley, Reading, MA, 1989.

[13] W.F. Tichy, R.L. Adams, and L. Holter. NLH/E: a natural-language help system. In *Proceedings of the 11th ICSE*, pages 364–374, Pittsburgh, PA, May 1989.

[14] C.J. van Rijsbergen. *Information Retrieval*. Butterworths, 2nd edition edition, 1979.

[15] R. Wilensky, Y. Arens, and D. Chin. Talking to UNIX in english: an overview of UC. *Communications of the ACM*, 27:574–593, 1984.

[16] M. Wood and I. Sommerville. An information retrieval system for software components. *SIGIR Forum*, 22(3,4):11–25, Spring/Summer 1988.

A Framework for Creating Environments

Axel Mahler and *Andreas Lampen*
axel@coma.cs.tu-berlin.de, andy@coma.cs.tu-berlin.de
Technische Universität Berlin
Sekr. FR 5-6
Franklinstraße 28/29
D–1000 Berlin 10, Germany

1. Introduction

The purpose of Software Engineering Environments (SEE) is to increase the efficiency of software production, and to enhance the quality of the developed software. The key concept in achieving this goal is a comprehensive support for the software development *process*.

In theory the effect of an environment would be to relieve the members of a project team from having to be aware of complex procedures and behavior protocols that are essential for the functioning of a project. The environment ensures automatically that these protocols are followed. Programmer productivity is increased because the programmer can better concentrate on the problem he/she is working on.

1.1. Software Engineering Environments must be adaptable

For some reason SEEs tend to not satisfy the expectations that are associated with them. Despite the recent "CASEmania" many integrated toolsystems and environments suffer from low acceptance or simply don't perform as expected. It has even been noted that environments eventually happen to be counterproductive.[5] Over the last decade substantial research has been (and still is) conducted to attack these problems and to generally improve the usefulness of SEEs.

While early SEEs were normally centered around a static model of the software development process (derived from a software development method) it has been learned that software development projects are highly dynamic processes, strongly dependent on the people involved, the size and complexity of the developed system, as well as external constraints such as customer relations. Today, it is understood that the actual process of developing software is more or less different for every project.[13] *SEEs must be able to cope with changing characteristics of the software development processes that have to be supported.*

What is needed is a customizable, highly flexible, and extensible SEE-frame/kernel. Some of the more obvious reasons for the necessity of SEE extensibility are:

- different projects need different kinds of support
- an environment should adapt to real (social) processes instead of forcing the teamwork structure to adapt to the environment
- fault-tolerance against environment design flaws
- support for environment evolution according to evolving requirements when a project proceeds (for example from implementation to maintenance phase)
- allowing *partial completeness* of an environment, implying delayed functional completion when suitable tools become available (rather than integrating insufficient tools "just to be complete")

During the last years, it has been widely accepted that good SEEs should be integrated, yet open and highly extensible.[6, 7, 11] Recent research has furthermore led to an increased awareness that in order to make the extension approach feasible, the extension of a SEE can be performed *rapidly*.

A good SEE shall combine the integrated look and feel of – for example – Smalltalk or Lisp systems with the general scope and applicability of tool collections as in the Unix environment. The initial establishment of a stable, basic environment infrastructure for a given project shall be a matter of *days*. To apply major changes/extensions (such as adding new classes of software objects to the system) shall be a matter of *hours*, and to apply minor changes (for example set user preferences, or fine tune certain object behavior) shall be a matter of *minutes*.

1.2. Recent approaches to environment integration

Many recent research projects have adopted the notion of *Software Object* as central for environment extension mechanisms.[2, 4, 15] The specifics of the software development process are captured by classify-

ing the pieces of information evolving in a project, and describing related *behavior* of these information objects (we will refer to them as *software objects*).

The object oriented paradigm is used to specify external properties of software objects, and to provide for formally defined access protocols to be followed when manipulating these objects. Use of type inheritance makes it possible to specify general functionality on an abstract level (*abstract superclasses*), making the functionality available for use/reuse in application specific class definitions (*specialized classes*). The specified functionality is turned into action by corresponding *process programs*, formal (and executable) descriptions of particular software engineering activities, enacted by a process program interpreter. These process programs are the *methods* associated with a given software object class.[3, 14]

Another interesting, yet entirely different, approach to environment integration has been taken with the Field environment.[10] Field integrates existing tools under a consistent graphical user-interface framework. Tools communicate via messages, representing information and commands intended for other tools. Messages are sent to a central environment message server which dispatches the messages to all tools possibly interested in them (*selective broadcasting*). The technique permits to integrate unrelated tools by adding a *message interface* to them. While this approach makes it possible to create moderately highly integrated environments from existing tools without a great deal of work, it requires some modifications to the tools' source code.

The most advanced approaches (for example [3]) are using knowledge based technology as main extension mechanism for SEEs. Today, we are moving towards a point where a SEE is basically a system that allows to store *knowledge* about how a particular project is working, and *functionality* that is intelligently applied according to the project specific work patterns stored in the knowledge base.

In the remainder of this paper we will introduce the STONE † project, a research project aiming at development of a generic SEE kernel especially suited for *environment education*. We will briefly discuss different techniques for environment integration and, more detailed, outline STONE's approach to *rapid* tool integration.

† STONE – a STructured and OpeN Environment, is a research project supported by the German Ministry of Research and Technology (BMFT) under grant ITS-8902E8. STONE is a joint project of nine institutions in the research and education domain of whole Germany. Participants are five research institutes and four universities.

2. Integration Techniques for a Generic Software Development Environment

STONE's principal agenda is to provide the technology for creating *integrated* and *open* software engineering environments. It is also a main goal of the project to support education and training of *environment engineers*. One of the reasons for launching the STONE project was the perception of a growing need for highly qualified people who are capable to build, alter, and maintain SEEs according to the ever changing needs of evolving software development projects. We consider this to be a crucial issue for increasing the acceptance of SEEs in real-world software development.

The intended SEE kernel shall be the seed for an environment that is open to extension and integration of new (i.e. dedicated) as well as existing ("off-the-shelf") tools. The kernel shall provide basic functionality, such as *object management*, means for *software process modeling*, and an adaptable *user interface*. Part of the STONE research is the investigation of various integration techniques that are applicable to the task of environment integration, namely fine-grain data-integration, coarse-grain data-integration, and user-interface integration. We refer to these approaches as

- *Integration by design*, i.e. the components are developed with integration as a design criteria
- *Integration by adaptation*, i.e. mutually independent, reusable components are adapted to each other by some sort of "glue", and
- *Integration by appearance*, i.e. different components – integrated or not – are presented in a uniform manner at user interface level.

In this paper, we shall have a brief look at STONE's integration-by-design strategy, and a comparatively more thorough look at the *Rapid Tool Integration Service* (RTIS), STONE's integration-by-adaptation technique. User-interface aspects will not be discussed here.

2.1. What means "Integration" ?

Generally, integration means creating a new, higher order *whole* from a couple of smaller pieces that is qualitatively more than the mere sum of its constituent parts. The classical example of an integrated system in the area of software development support is a programming environment. Integrated programming environments consist "of a number of tools that assist a programmer in the program construction, test, and validation process. These tools include editors, debuggers, compilers, pretty-printers, test-case generators, various kinds of analysis aids, and so on. Many of these tools operate on some intermediate representation of the program: a form that is below the level of the source text"[12]. To achieve a high degree of semantic interaction among these tools, requires fine-grain data integra-

tion atop the intermediate program representation. Characteristical for this integration approach is a comprehensive design process that involves well coordinated design of the data-model, and the processes (tools, components) that employ this data-model. A serious drawback of this integration approach is that environments of this kind are

- expensive to build
- closed for integration of external tools that weren't build with respect to the integrated data-model
- hard to alter; changes in the data-model may imply changes of some or all tools.

However, highly integrated programming environments are an essential part of any effective SEE. The STONE fine-grain data integration concept facilitates building of highly integrated sets of tools by using a strongly object-oriented approach to data modeling[16]. Integration basis is an object-oriented data model with all data stored in the database. An integrated environment data model is accessible via the data-base's metaschema and thus can be used to build specialized versions of tools without altering existing definitions in the data model. This corresponds to the *open-closed-principle* formulated by Meyer[9]. It is also possible to add new tools to an existing programming environment because all structural data is readily accessible. The data management system derives structure definitions and access routines in a host language (currently C++) from the data model. All tools to be integrated with respect to a given data model have to include the generated definitions and routines in order to be able to access data in the database. According to the problems described above, this approach is only feasible for construction of (parts of) environments with a relatively stable data model. Although we expect substantial improvements with respect to the latter two of the above noted points, it remains the problem that tools with a high degree of semantic interrelation have to be custom-made. We argue that this kind of fine-grain data modeling as *sole* basis for constructing software engineering environments is not sufficient, as it would yield a much too expensive and inert way to building support environments.

3. Rapid Environment Construction

STONE's integration-by-adaptation mechanism, described in this section, aims at *rapid* creation of a basic support environment. We consider the UNIX† environment to be an ideal basis for this enterprise. Unix provides a rich set of tools for all imaginable (basic) tasks. We want to employ this tool base as a library of building-blocks (in the sense of *reusable procedures*) for higher order, integrated *functionality* that will be associated with *software objects*. STONE's rapid tool integration service (RTIS) provides a comprehensive framework for embedding Unix tools into a coherent project support environment. Users of such an environment need neither be aware of the existence of particular Unix tools, nor of the peculiarities of how to properly invoke them.

The concept is based on software objects modeled according to the file notion of Unix. However, software objects are based on a consistent, general purpose type system. The typed software object abstraction is built on top of a general attribution mechanism for Unix files (and versions) described in [8].

The type system allows to describe *classes* of software objects that are more than just files. For example, a C-language source object would not just be an ordinary file that happens to contain C source code but would be associated with properties (attributes and methods) that are characteristical for C code modules. The type system is defined in an object oriented specification language called *CHieF* (Class Hierarchy Definition Facility), featuring multiple inheritance, generic classes, method overloading, dynamic identification, and easy schema modification. This specification mechanism allows to easily describe the properties of the various software objects encountered in the software development process, such as relations to other software objects, or dependencies. The specified methods for a given object class are implemented by Unix-tools or a combination of Unix-tools (shell scripts).

The type system is enacted by *OShell*, an object oriented command interpreter resembling the Unix Shell. OShell allows to send objects messages that trigger invocations of object specific methods. In that the system is not unlike the *osh* idea described in [1]. A similar concept is present in Odin's [2] request language, but lacks a sufficient inheritance mechanism.

The type system and the object shell allow to capture the external behavior of Unix tool processes, such as *vi*, *awk*, or *cc* if they are described as methods. Such behavior would for example be the creation of certain new objects (files) or the type of data that will be read from standard input or sent to standard output. This in turn makes it possible to construct type-safe *pipelines* within the object shell.

Adding support for a new kind of software object basically consists of defining the respective class, possibly using predefined functionality (such as version control) via the inheritance mechanism, and implementing the corresponding methods in form of calls to Unix tools (e.g. version control programs), or (O)Shell scripts that are also stored in the object base as special software objects of type "Method". By applying this idea, we are

† UNIX is a trademark of AT&T Bell Laboratories.

able to amalgamate the prototyping power of Unix-typical tool combination (shell scripts, pipelines) with a formally integrated behavior of the environment.

While OShell itself is not a graphical user interface, it can be used to interact with a graphical environment *shell-tool*. In fact, the object shell represents an *abstraction* that has a well defined syntactical interface, suitable for formally communicating with a user interface process that presents the concepts of the object shell in a graphical way on a workstation. A schematic overview about the architecture of the rapid tool integration facility is given in the following Figure.

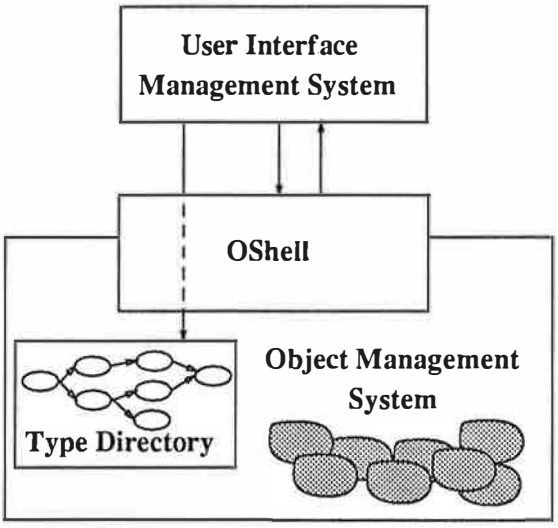


Fig. 3.1. Rapid tool integration in STONE

References

1. Timothy A. Budd, "The Design of an Object Oriented Command Interpreter," *Software – Practice and Experience*, vol. 19, no. 1, pp. 35-51, January 1989.
2. Geoffrey M. Clemm, "The Odin System: An Object Manager for Extensible Software Environments," *CU-CS-314-86*, The University of Colorado, Boulder, Colorado, February 1986.
3. Geoffrey M. Clemm, "The Workshop System – A Practical Knowledge-Based Software Environment," *Software Engineering Notes, Vol. 13, No. 5*, pp. 55-64, ACM Press, Boston, Mass., November 1988.
4. William Courington, "The Network Software Environment," *A Sun Technical Report*, Sun Microsystems, Inc., Mountain View, CA., May 1989.
5. Peter F. Elzer, "Management von Softwareprojekten," *Informatik Spektrum*, vol. 12, no. 4, pp. 181-198, Springer Verlag, Berlin, August 1989.
6. A. Goldberg, *Smalltalk-80, The Interactive Programming Environment*, Addison Wesley Publ. Company, Reading, Menlo Park, London, Amsterdam, 1984.
7. A. Nico Haberman and David Notkin, "Gandalf: Software Development Environments," *IEEE Transactions on Software Engineering*, vol. 12, no. 12, pp. 1117-1128, December 1986.
8. Axel Mahler and Andreas Lampen, "An Integrated Toolset for Engineering Software Configurations," *Software Engineering Notes, Vol. 13, No. 5*, pp. 191-200, ACM Press, Boston, Mass., November 1988.
9. Bertrand Meyer, *Object-oriented Software Construction*, Prentice Hall International, 1988.
10. Steven P. Reiss, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, vol. 7, no. 4, pp. 57-66, IEEE Computer Society, July 1990.
11. G. Snelting, "Experiences with the PSG – Programming System Generator," *Lecture Notes in Computer Science*, vol. 186, no. 2, pp. 148-162, Springer Verlag, Berlin, March 1985.
12. Richard Snodgrass, *The Interface Description Language: Definition and Use*, Computer Science Press, ISBN 0-7167-8198-0, Rockville, MD, 1989.
13. Vic Stenning, "On the Role of an Environment," *Proceedings of the 9th International Conference on Software Engineering*, pp. 30-34, IEEE, Monterey, California, March 1987.
14. Richard N. Taylor, Richard W. Selby, Michael Young, Frank C. Belz, Lori A. Clarke, Jack C. Wileden, Leon Osterweil, and Alex L. Wolf, "Foundations for the Arcadia Environment Architecture," *Software Engineering Notes, Vol. 13, No. 5*, pp. 1-13, ACM Press, Boston, Mass., November 1988.
15. Walter F. Tichy, "Tools for Software Configuration Management," *Proceedings of the International Workshop on Software Version and Configuration Control*, pp. 1-20, German Chapter of the ACM, Grassau, FRG, January 1988.
16. Jürgen Uhl, Bernhard Schiefer, Emil Sekerinski, Simone Rehm, Thomas Raupp, Michael Ranft, Richard Längle, and Karol Abramowicz, "The Object Management System of STONE - SOS Release 1.0 -," *STONE Technical Report FZI.001.2*, Forschungszentrum Informatik, Karlsruhe, April 1990.

Emeraude, a Unix-based implementation of PCTE

Régis Minot, GIE Emeraude
Bull, 58 route de Versailles
78430, Louveciennes - France

1. Introduction

PCTE (Portable Common Tool Environment) is gaining more and more acceptance in the domain of frameworks for the development of Integrated Project Support Environments. This is the result of an international initiative from both major computer manufacturers, companies which are facing the problems of developping software in the large and programmes from the CEC (such as ESPRIT and EUREKA) or the IEPG (Independent European Programme Group). This is also the result of the existence of an industrial implementation of the PCTE framework. This implementation, Emeraude, is based on the Unix operating system and is becoming available on several major workstations of the market. This paper provides a summary of the functionalities offered by the PCTE interface and shows how a Unix based implementation contributes to the portability of the PCTE implementation itself.

2. The PCTE functionalities

2.1 The basis: a well defined specification

There are currently two versions of PCTE:

- The PCTE 1.5 [8] specifications of PCTE which itself is a revision of the former PCTE 1.4 [7] specifications, constitutes the basis of the currently available implementations. The Emeraude product [2] has been the first implementation of PCTE 1.4. In its current version, called V12, Emeraude implements PCTE 1.5 on top of Unix.
- The PCTE+ specifications [9,1] which were developped on demand of the IEPG, constitute the basis for the current standardisation work of the ECMA/TC33 comittee as well as the basis of an assessment programme which includes two implementations of this interface. The ECMA/TC33 comittee which includes most of the major computer manufacturers, has produced, from PCTE+, an ECMA PCTE Standard proposal which will be voted at the end of this year. The form of the standard consists of an abstract syntactic and semantic specification [10], completed by two bindings for C and Ada.

An important goal in the evolution from PCTE 1.5 to ECMA PCTE has been to preserve the developers investments by keeping an upwards compatibility of concepts. Furthermore, the Emeraude V12 implementation which is the subject of this article already includes some of the new functionalities of the proposed ECMA standard and therefore constitutes a meaningful step in the building of a complete Integrated Project Support Environment.

---

* Unix is a trademark of AT&T

According to the CASE reference model [4,11] established by ECMA, the functionalities of PCTE can be analysed in three directions:

- control integration
- user interface integration
- data integration

2.2 Control Integration

PCTE has acknowledged the existence of a good level of control integration services in the Unix operating system: therefore the PCTE execution model and communication model are widely derived from the Unix model. The main extensions consist in a systematic approach to the transparent distribution of these functionalities over a network of hosts ranging from diskless workstations to mainframes acting as servers.

PCTE has nevertheless extended the process management facilities to support the notion of *static context*, *interpreter* and *execution class*:

- A static context consists of a program with additional properties. A process is started by activation of a static context. A static context can be either interpretable or executable.
- When a static context is interpretable, it is linked to another static context which constitutes the interpreter of the text of the former static context. The invoker of a static context does not need to care about the activation of the interpreter.
- A static context can be linked to an execution class object which indicates what range of stations are able or allowed to execute processes started from it. Here again, the invoker does not need to care about the choice of the station which is made transparently by the system.

Another extension to Unix is the notion of named message queue. PCTE message queue are named objects and can be used in a multiplexed way: consumers can select messages according to a given message key. It is also possible top examine the contents of amessage queue without necessarily removing messages from the queue.

2.3 User Interface Integration

Several advanced works have been made in the area of user interfaces, especially with X/11 [12] which is becoming a de facto standard. This fact has been acknowledged by ECMA which has adopted X/11 as its basic portability layer of the user interface functionalities for PCTE. This does not preclude tool developers to use higher level toolkits such as OSF MOTIF [ ], since MOTIF is portable on top of X/11.

2.4 Data Integration

Data integration is the domain where the lack of adequate functionalities is crucial in most of operating systems. It is also the domain where PCTE, through its Object Management System [6,5], has placed most of its efforts.

Higher levels allowing objects to be managed are clearly necessary. The Object Management system of PCTE provides such functionalities. It is based on an extension of the entity relationship [7] model with features of object oriented models such as type inheritance and several characteristics suited to solve typical problems of software engineering.

The basic concepts are those of *objects*, *links* and *attributes*. Objects may represent the data which are used to manage the projects (e.g. the development tools, the project plans, the tests, the users and the resources of the environment) as well as the data which are produced by the project (e.g. the modules of a target application). These data can be structured in atomic or composite objects.

Composite objects have components which are also objects but operations are provided to manage such composite objects as a whole.

The objects are interrelated by relationships which are pairs of *links*. Each link can be navigated as part of a sequence of links or pathname. The access to the object base follows the same principles as the access to a Unix file-system but allows relationships to be established between all types of objects (including files) and without an hierarchy structure. One to one, one to many or many to many relationships can be modelled.

Finally objects and links can have properties, or *attributes*.

The execution model allows processes to access the object base with controls on both data integrity and access concurrency. The data integrity is provided by means of *schema definition sets* describing the data structure of the object base in a modular and extensible way. A schema definition set is a set of object types, attribute types and link types which define common properties of sets of instances.

The access concurrency is controlled by the notion of *activity*: activities can offer different levels of protection ranging from non protected activities to transactions. Transactions, like the other classes of activities, can be nested.

Finally, the whole object base is transparently distributed on the various volumes managed by the stations of a given PCTE installation. The users and tools see a single logical object base and can ignore the actual physical distribution of the data.

3. Emeraude V12

The goal of PCTE is to provide a platform (or framework) offering both portability and common integration services to IPSE builders and tool vendors. Emeraude is an industrial product which implements this interface (currently the PCTE 1.5 one). An additional goal of the Emeraude V12 implementation of PCTE is to be itself portable (as much as possible) on top of Unix.

The Emeraude architecture consists of one server process per station and one object server per station which has disks. These servers are exchanging data by means of the TCP/IP protocole.

The PCTE processes are client processes for their local station server. A PCTE process is developed like a Unix process but needs to be linked with a library which ensures the interfacing of the PCTE operation calls with a remote procedure call to the server.

The Emeraude V12 environment offers various services such as objects, links and attributes management tools, composite objects versions management, user assistance to invoke tools through menus, schema design and tools to manage the metabase (the database of object types) as well as common programming services provided as libraries.

The first release of Emeraude V12 is currently distributed over a local area network of homogeneous stations. The second release will support the heterogeneity. Additional administration tools are provided to manage the resources of the installation (users, groups, stations, volumes) and to replicate the critical administration objects in order to allow work to be done in a non connected state (i.e. isolation of a station from the rest of the network).

Finally, the integration of development tools can be made at two levels:

- the first one, called *integration*, requires an analysis of the data manipulated by the tool and a modelling of these data in the object base: it is mainly used for new tools but can be also applied as modifications to the source of existing tools,

- the second one, called *encapsulation*, allows a Unix binary tool to be invoked on data of the object base, either by copy from the object base to the Unix file system or by sharing.

Emeraude provides a guide to assist the tool developers and integrators in the first approach. The second approach is also possible and is based on access sharing between the object base and the native Unix file systems.

4. Unix as a basic platform to port Emeraude

Emeraude V12 uses basic system calls from the Unix System V interface, together with functionalities of sockets, NFS and TCP/IP. This set of basic functionalities appears to be available on most of Unix stations and therefore constitutes a valid portability basis for Emeraude itself. Starting from the initial version developed on Sun stations, Emeraude V12 is currently ported on Bull DPX/2, DEC 3100, Hewlett Packard HP 9000 and IBM RS 6000.

This is not a strict portability since some minor differences exist between the Unix systems. We also do not consider Unix as a sufficient level of integration for tools (especially in the area of data integration).However, the experience of Emeraude V12 is showing that Unix is a very good basis to support an higher level portability and integration interface such as the one offered by PCTE.

References

1. Boudier G., Gallo F., Minot R., Thomas M.I., "An Overview of PCTE and PCTE+", Proceedings of the Third ACM Symposium on Software Development Environments, Boston, November 1988, in *ACM SIGSOFT Software Engineering Notes*

2. Campbell I., "Emeraude Portable Common Tol Environment", Information and Software Technology, Volume 30, No 4, May 1988.

3. Chen P.P., "The Entity-Relationship Model: towards a unified view of data",

4. Crispin R., Earl A., ECMA TC33-TGRM, "A Reference Model for CASE Environments", PCTE Newsletter Number 3, February 1990.

5. Gallo F., Minot R., Thomas M.I., "The Object Management System of PCTE as a Software Engineering Database Management System", Proceedings of the Second ACM Symposium on Practical Software Development Environments, Palo Alto, *ACM Sigplan Notices* , Vol 22, No 1, Jan. 1987.

6. Minot R., Gallo F., "The Object Management System of PCTE as a Software Engineering Database Management System", 2nd Future APSE Environment workshop, Invited paper, Sept. 86.

7. PCTE 1.4
   Bull, GEC, ICL, Nixdorf, Olivetti,Siemens, "PCTE Functional Specifications, Version 1.4, C Binding" , Sept 1986.

8. PCTE 1.5
   PCTE Interface Management Board (CEC), "PCTE Functional Specifications, Version 1.5, C Volume 1 (OMS) + C Volume 2 (UI)", Nov 1988.

9. PCTE +
   GIE Emeraude, Selenia, Software Sciences Ltd, "PCTE+ Ada and C Functional Specifications", Issue 3, Nov. 1988.

10. ECMA PCTE
    ECMA, European Computers Manufacturers Association, Abstract Specification, Final Draft, November 1990, Submitted as ECMA Standard at General Assembly of December 1990.

11. ECMA CASE Reference Model
    ECMA, European Computers Manufacturers Assciation, A Reference Model for Computer Assisted Software Engineering Environment Frameworks, Final draft, submitted for adoption as ECMA Technical Report, Version 4.0, August 17, 1990.

12. X/Window
    XLib - C Language X Interface Protocol, Version 11, M.I.T., June 1987.

Position Paper

NeTS: Computer Network Analysis Tool

Hideo Nakano
Osaka University
nakano@oucom.osaka-u.ac.jp

Increasing the power of UNIX-based workstation, it becomes popular to construct a software development environment on many computers that are connected by networks. Software Engineers Association(SEA) in Japan has held Software Development Environment Workshop since 1985. I became a program chairperson on 1988 and a chairperson on distributed environment session.

On this international workshop, I want to discuss an efficient UNIX based network that supports best software development environment. Followings are brief summary of network simulator we are now designing and implementing.

NeTS

1. Introduction

We now design and implement a network simulator mainly based on UNIX Workstations and Local Area Networks. Software development environments are used in a large network including a hundreds of or thousands of workstations in many companies and universities. So the design of optimal network configuration is important problem. But the precise analysis of traffic in such network is difficult problem in a theoretical sense.

And also there are typical jobs such as remote login, file transfer, and network file system in these network environments. So the analysis must reflect for these typical jobs. In our network simulator, user(network constructor or manager) can assigns jobs to each workstations to evaluate the real traffic.

One of the reasons that we can construct visualized network simulator depends on the familiarity of Graphical User Interface Facilities. And also the terms of scientific visualization and algorithm animation become popular.

2. Network Simulator

Our network simulator, called NeTS, are designed to handle a hundreds of or thousands of computers connected by networks. The purpose of NeTS is to analyze the traffic in these practical computer network systems. There are three categories of elements consisting networks in our simulator. First one is computer. We include Mainframe computers, workstations, and also personal computers. UNIX system with TCP/IP is a main target for our analysis. Second elements is network including EtherNet, TokenRing, serial line, and public line and so on. Final one is network device such as router, bridge, and communication server. User can easily construct a desired network configuration using icon- and menu- based graphical user interface.

3. Job and routine

To simulate practical network, we consider the concept of job and routine. Remote login session is one typical job. The other typical job is single file transfer session. Another type of network job is network file system.

Usually computer network works everyday and every time. But almost every user works on his calendar. So let a routine to be a set of jobs in one day. For example, remote login session from workstation 1(WS1) to workstation 2 (WS2) in the daytime, 20KB file transfer from WS2 to WS1 in the morning, and 50KB file transfer from WS2 to WS1 in the evening consist one routine.

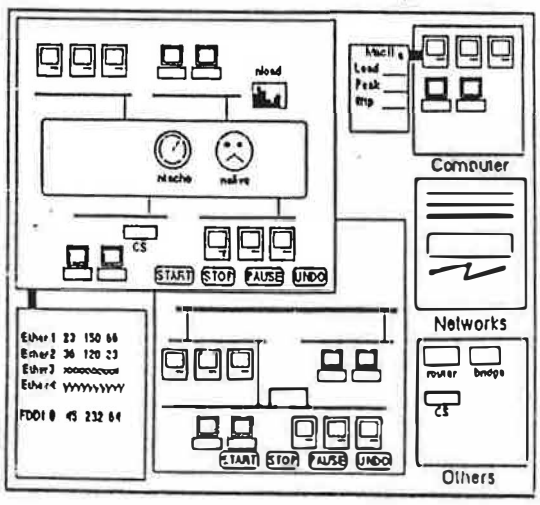
4. Simulation algorithm

There are many researches that explore the performance of each network such as Ether net by queueing theory. But it seems to be difficult to analyze the total performance of practical workstation-based network. So we adopt an algorithm that simulates the whole network according to the individual performance of each computer, network, and network device.

5. Displaying performance

There are some performance display tools such as meter, gauge, graph, and so on. In this simulator, user can select among them. Fig.1 is a basic idea of NeTS.

Fig.1 Sketch of Idea



Experiences of a Transition to SunOS™ Development under the NSE™

Tadd Ottman (tadd@widor.Eng.Sun.COM)

operating system development technology
Sun Microsystems, Inc.

Abstract

This paper presents a few of our experiences within Sun Microsystems in getting operating system development under the Network Software Environment (NSE). It describes the problems that motivated the transition to the NSE, the timing of the transition, the preparations made, the methodology used, a survey of our NSE users, players' viewpoints on the NSE, and some lessons learned. This gives an overview of the difficulties of adapting a large body of complicated source code to the NSE's requirements and adapting a large number of operating system engineers' habits to the NSE's requirements.

Introduction

In the first half of 1990, we in the operating system (OS) group at Sun have adopted the Network Software Environment (NSE)[1] to manage the development of our source and the tracking and building of our releases. Coincident with this adoption is a host of other adjustments to the way we manage and build our source. Few of these adjustments were required by the NSE; rather, some were made possible by the NSE. Members of our group have experienced the adjustments and the NSE as one big change in the way we do our development, so this paper's description will extend beyond the topic of the NSE alone.

As an organization, we had many old problems with our former way of managing OS development. These were widely acknowledged within the group. We knew some new measures would have to be taken to address the problems. In general, those most familiar with the old problems and closest to them have shown and are continuing to show the greatest willingness to invest in new solutions and to accept the short-term cost of accommodating the new tools. This makes it important to know the problems we faced and makes it interesting to see the viewpoints of persons playing differing roles in the project.

The Motivating Old Problems

A convenient focus of attention on the old problems is the build process. A prominent person in our organization wrote a widely circulated internal paper on "Fixing the Build." It listed a litany of problems, many interrelated. These included:

- inadequate or nonexistent testing of contributed source and bug fixes,
- difficulty in setting up buildable snapshots of the source,
- wasteful requirements for a dedicated "build" machine for each target architecture,

---

1. Evan W. Adams, Masahiro Honda, and Terrence C. Miller, "Object Management in a CASE Environment", *11th International Conference on Software Engineering*, pages 154-163, Pittsburgh, May, 1989.

- lack of a clean top-down build, causing people responsible for builds to have to follow a long list of detailed steps to complete a build with ad hoc tools,
- the inability to isolate many blocks of source from the whole for independent building and testing (nearly everything had to be built together from scratch as a totality),
- a requirement that each build machine run the version of the OS that it was attempting to build (posing great hardships during early unreliable versions of the OS),
- great inconsistency in makefile style and content (adding to the cost of maintenance and the build engineer's learning curve),
- and the almost constant need for a senior engineer to monitor the build process and explicate the integration problems it uncovered.

Other problems not pertaining to the build process, per se, were noted. These included:

- poor control over integration (what was getting into the source base, when, and by whom),
- occasional losses of SCCS deltas as version files got overwritten,
- poor tracking of the collections of files involved in the fixing of specified bugs or the implementation of specified features,
- and merge difficulties caused by disparate development groups working on separate source bases.

The Timing of the Transition

The timing of the decision to fix these problems was determined by the fact that we were switching to an entirely new source base for our next release of the OS, namely System V Release 4 from AT&T. This entailed a port to SPARC from 3B2 and a thorough restructuring of makefiles and the build process, anyway. The marginal cost of adding NSE compatible makefiles and bootstrapping the source into the NSE was thought to be minimal at this juncture. Moreover, few engineers would be working with the source initially, implying that few would have to learn about the NSE early and others could be introduced to the NSE in phases as the project grew.

Preparations Made

Although we began our use of the NSE on the mainstream of OS development this year, we made experimental use of the NSE much earlier. Source for older releases of the OS was bootstrapped into the NSE to gain experience with the NSE, to evaluate the consumption of disk space and other machine resources, and to identify incompatible makefile practices. (Bootstrap verifies and uses makefile targets. It copies files, accepts file revision histories, and creates component hierarchies and tables of symbolic links.)

This prototyping work joined with "fix the build" suggestions to form the basis for new makefile standards and guidelines. In this sense, the NSE acted as a catalyst for change (and a scapegoat for problems). While the makefile guidelines were written ostensibly to implement and promote NSE compatibility (permitting bootstrapping), many other techniques and policies were included to fix the build. The NSE's support for variants allowed the makefiles to assume a simpler design; files derived in the course of a build may occupy the same name space for all architectures because the NSE stores and manages them in isolated filesystem layers, using the translucent file service (TFS).[2] The NSE reliably sets variant-specific variables in the shell environment, so these may be used in makefiles dependably. The language tools used in the course of a build may be invisible to the makefiles yet differ according to variant when used appropriately in NSE execsets. (Execsets are sets of executables and other build resources in a filesystem made available to the user automatically when an associated environment is activated. There may be several members of an execset corresponding to variants in an environment.)

Prototyping made it very clear that the NSE of the day could not handle what we call source variants, e.g., architecture-specific groupings of source. The operating system, of course, often calls for writing and

---

2. David Hendricks "The Translucent File Service", *Proceedings of the European Unix Systems User Group Autumn 1988 Conference*, October 1988.

maintaining machine-specific source code in well-defined places such as subdirectories. This made it necessary for the OS group to use a new version of the NSE containing an added twist to the variant functionality, supporting what is called "target variants." (The dependencies the NSE associates with specified makefile targets are recorded and tracked in a variant-specific manner, allowing differing name spaces across variants for sets of sources and dependency graphs.) The transition to doing software development under the NSE has been made more painful and more interesting by the requirement of using alpha-quality new functionality in the NSE. However, it is generally felt that the new source base presented a unique opportunity to migrate to a new way of doing our development, so we could not wait for the NSE to stabilize fully.

Further preparations included careful planning of our hierarchy of environments and of the roles played by each in development, testing, and release. At one level of the hierarchy, environments were associated with groups in our organization responsible for implementating various functional areas of the operating system. This tied the NSE to new organizational processes for project management which were being set in place at the same time.

We wrote and presented internal classes to members of the OS group on the new software development process, use of the NSE, our environment hierarchy, release plans, makefile guidelines, and more. To accompany the classes, we devised some lab exercises. This forced us to think through our ideas and gave us some experience in explaining them. To supplement the classes, we wrote developer guidelines, integration guidelines, integration rules, and instructions for installing the NSE and creating execsets. We even produced a videotape.

The Methodology of the Transition

The physical transition to managing source and builds under the NSE was simple, relative to the social transition. We maintained a centralized source hierarchy with SCCS history on one machine. New makefiles were written and tested on this machine. As modules became ready, we took snapshots of planned components of this source, placed the snapshots on a separate machine, tested the snapshots, and bootstrapped the snapshots into the NSE. As components "arrived" in the NSE, they were invalidated back in the SCCS hierarchy. The product of our routine builds consisted of the combination of the build inside the NSE and the build outside. Bootstrapping has doubled as the enforcer of makefile standards during each component's testing phase.

A Survey of Our NSE Users

The choice of the NSE for management of software development was made in the long-term interest of improving the design and implementation of the NSE while giving the OS group necessary functionality through tools it could control more than third-party solutions. It is too early for us to assess the merits of the choice from a long-term perspective, but a recent internal survey indicated that the NSE-user community of OS engineers was evenly divided on whether the NSE would be worthwhile for the group in the long run. With due consideration of a number of factors, this, I believe, was a positive result.

Before presenting survey perspectives, it is good to consider factors that influenced the survey. The version of NSE being used is of alpha quality, and our OS source is stressing the newest functions. Many of the advantages of the NSE accrue to integration and test engineers and the project management while posing a burden or a perceived loss of productivity for the development engineers comprising the majority of the survey sample. These advantages include the coordination of integration, the insulation of individual environments from the changes being made by others, the provision of an insulated test space prior to integration of new source, and the target-consistency checking done during all inter-environment exchanges of source. Our organization's use of the NSE is yet too immature to have progressed through several minor releases under concurrent development, so few members of the group can now see or appreciate the difference the NSE is beginning to make, but many recall their experiences of learning this new and complex product and making mistakes with it.

Player Viewpoints

All the players in our organization have opinions about our transition to the NSE; the stress and newness have engendered strong viewpoints, positive and negative. Here is my understanding of a representative few.

The typical developer, with a memory of how things used to work, complains about a frustrating loss of productivity. He or she sometimes praises the NSE for the consistency and completeness of source one can acquire for specific software components, or for the ease of merging two branches of development and resolving conflicts that result. The complaints concern dependence on more hosts on the network, slower builds on TFS in activated environments, cryptic or misleading error messages, and a time-consuming reliance on NSE gurus when things go wrong with the alpha version. Because the NSE provides a reasonable environment for testing before integrating, the developer has little reason not to carry out more steps in the process of delivering changes to the source base (and our new procedures now require such testing). Setting up and running tests is seen by many as a loss of their productivity.

The integration engineer's view is mixed. There is much more control over what gets integrated and in which order, but integration is serial while the development is parallel. Better locking and access control for environments are desired. The typical NSE operation on a large integration environment is slow and expensive. Recovering and restarting operations are often painful. However, the quality of the contents of the integration environment is higher than ever. Incremental builds are easy. The insulation of one developer's changes from the next makes it much easier to trace responsibility for introduced problems and conflicts; hence it is easier to maintain quality.

The project manager appreciates the clarity and separation of functional groupings of project tasks afforded by environments but suffers from the current alpha NSE's lack of robustness. The whole organization is learning from the dependencies that are revealed between functional groups at integration time. The developers' problems with running tests and with learning how to use the NSE effectively introduce new degrees of variability and risk in task management. To some extent this is balanced by a new dependability in the process of building and issuing any specific release. Automatic notification from the NSE of specified events occurring within environments of interest is a big plus to the project manager.

The poor system administrator sees little gain except perhaps a new measure of job security. Machines running the NSE are special, especially those hosting high-level integration environments. Downtime is bad any time. Many more people need be consulted for scheduling downtime. Shared NSE environments cause filesystem interdependencies and thus a need to time-order partition backups in a new way, often entailing coordination with other system support organizations. Because of the directory structure of branch root directories (where environments are stored on disk), the time needed to back up or restore environments is longer than usual. Furthermore, tar usually fails with branch root directories, limiting solutions for moving or archiving environments. The NSE imposes new NIS (YP) map requirements.

These are the problems an administrator would face without being asked to wear a new hat as NSE administrator as well. Then, more problems surface. These include learning to reconstruct damaged environments, ensuring inter-environment consistency, managing disk space with NSE marking and garbage collection, moving environments, renaming environments, creating and updating execsets, checking and repairing TFS file information, installing and upgrading the NSE, and more.

The manager's view is reflected in the decision to adopt the NSE in the first place. The new problems it presents can be addressed through new policies, more education, more machine resources, and technical improvements to the NSE. These are generally easier to manage than the old problems the NSE helps solve. The greater sense of process and control gives the organization direction. The smooth builds and added testing point to real gains in software quality. Most of the factors contributing to schedule delays are thought to be short-term. Assessed in light of both functional productivity and quality, the group may already be benefitting from the NSE and new procedures.

The corporate view sees the NSE and the OS as two groups and two products that can and must be mutually beneficial. The OS is a trial by fire for the NSE, while the NSE is maturing the OS software development process. The whole experience is building a basis for future solutions to the on-going problems encountered.

The NSE group's product-support view mixes surprise, satisfaction, and weariness. Their surprise emanates from the realization that target variants are harder to implement well than was at first thought. Their satisfaction stems from knowing the product has come a long way: it is supporting many users with a large and complicated source base and satisfying many of the demands placed on it. Their weariness comes from working hard for so many months supporting so many users with so many questions and bugs with the alpha version. (The number of bugs adds to their surprise.)

Things Learned

Perhaps from these views, you can infer many of the things we have learned. We have a lesson on learning itself. Any new methodology entails a learning curve. This places a burden on someone to provide education. We attempted to teach through our videotape, classes, labs, and documents. Looking back, we now see that we presented too much material at once, and we presented it so early that many engineers forgot it for lack of use. The lab exercises were ignored, so real environments became the learning environments. With something as complicated as the NSE, yet in other respects as familiar as anything else in UNIX† ®, new users easily get themselves into deep trouble without having a clue that they need to know much more before going beyond a certain step. Since such trouble situations are expensive to support, we now think it probably would have been less costly to provide continual "hand-holding" NSE assistance during the first encounters each engineer had with the NSE. The trade-off is akin to preventative medicine.

We have learned that parallel development is one thing, and parallel integration is another. Serial integration has become quite a bottleneck for us. The high degree of parallelization in development with the NSE has teased us into wanting at least a limited degree of parallelization in the integration process through appropriate locking.

We have learned that some characteristics of the NSE have been more costly to us than we expected. The bootstrapping process is one. For us, the requirement of having all source buildable from the top down with compatible makefiles was unrealistic. We had to move source into the NSE gradually and incrementally as it became ready. The serial nature of the work caused a bottleneck. The delays permitted incompatibilities to evolve between sources inside the NSE and sources outside. Further delays were caused by the strictness and span of makefile requirements. A few subtle incompatibilities took much time to discover, correct, document, and avoid.

We have found that automatic and dynamic updating of NSE targets with dependency checking is an asset, but it has a price. If targets are unavoidably large, as in the case of the kernel, then presumably simple operations take a long time to complete while many files are checked and database entries consulted. At some low levels of the environment hierarchy, we wish we could turn this feature off. Encouragingly, we are learning new ways to organize our source into more sensible units.

Conclusions

In most important ways it is too soon to draw conclusions from our transition into use of the NSE. It is clearly a powerful tool that takes time and energy to master. A phased approach to its adoption is best, however possible. The alpha version we now use cuts into developer productivity with its lack of robustness, but as it improves we are seeing benefits from parallel and insulated development, consistent targets, controlled integration, partially automated merges, conflict detection, event notifications, and improved environments for testing. The rewards will be greater as we master the tool.

† UNIX is a trademark of Bell Laboratories.

WhiteCap - A C/C++ Development Environment

David R. Reed

Saber Software Incorporated
Cambridge, MA

WhiteCap is an integrated program development environment for the C++ language. It is unique in that it effectively supports programming and debugging at multiple levels of abstraction. C++ is nominally a superset of C which provides very desirable upward compatibility. However this forces the implementors of C++ debuggers to deal with a run_time model that is more complex than C's and one that is much more undisciplined than for a strictly object-oriented language. Effective debugging in this model calls for a new combination of run-time checks. WhiteCap takes extensive C language debugging technology, generalizes it to cover C++, and adds powerful object-oriented run-time checking. The environment consists of an interpreter, an incremental linker, an interactive workspace/debugger, a set of browsers, and other tools.

The interpreter executes both AT&T Release 2.0 C++ and K&R C with ANSI extensions. Interpreted source may be loaded along with objects generated by Cfront and standard C compilers. Loading, unloading, and reloading of source and objects is performed by a special incremental linker, drastically reducing the time of the change/compile/link/test cycle.

WhiteCap implements two levels of run-time checking for code that is executed interpretively. One level of checking verifies that operations on low-level, built-in types (such as char and int) are executed on valid operands. A higher level of checking verifies that operations on class instances are valid. This level of checking is accomplished primarily by verifying that manipulations of address values of class instances are consistent with the type expected at the designated memory addresses. This level of checking is made possible by the interpreted execution facility, which retains more information about the class types declared in the user program than most standalone debuggers.

An interactive workspace allows direct interaction in either C++ or C. Any valid C++ or C declaration, statement, or expression may be entered into the workspace for immediate execution. This supports "bottom_up" testing of new source by calling it directly, without a separate test driver. At any one time the workspace is either in C++ or C mode. Mode is determined by either execution context (break level) or user selection. If at a break level, statements and expressions are interpreted in the scope of the break location, allowing flexible debugging in the source language. In addition to source interaction, an extensive set of debugging commands can be invoked in the workspace.

The user interface for WhiteCap is based on the X Window System. It provides several browsers for examining the program and its state of execution. The display browser provides a view of the user program's data structures. The user can name individual objects to be displayed as separate graphical elements. Additional objects that are pointed to by components of the original object can be selected for expansion and are visually linked to the original object. The amount of detail shown for each object can be controlled on a class-by-class basis. For example, classes under development can have all members displayed by default, while complete, stable classes may have only public members shown. A cross-reference browser shows the dependencies between program elements. References are not determined lexically, rather they are derived from the incremental linkers database of inter- and intra-module references. This display can show the references to and from any individual object, such as a member function. A class browser provides an easy to use tool for scanning the classes that are incorporated into a program. It features a "schematic" or simplified format for display of the contents of a class, with the level of detail controlled by the user.

NGCR Project Support Environment Standards

Carl Schmiedekamp
Code 7033
Naval Air Development Center
Warminster, PA 18974

(215)441-1779 , FAX (215)441-3225
(carls@nadc.navy.mil)

The Project Support Environment Standards Working Group (PSESWG) of the U.S. Navy's Next Generation Computer Resources (NGCR) project is planned to begin meeting in early 1991. Participation is open to anyone from Industry, Government or Academia. The PSESWG does not intend to build a PSE; rather the plan is to assemble a set of commercial-based interface standards from which Navy PSEs in the future will be built. No decision has been made as to which or even if there will be an operating system standard for Navy Project Support Environments. However it seems clear at this point that a decision will have to be made and that the IEEE 1003 (Posix) family of standards will have to be considered.

The U.S. Navy has embarked on a new computing resources standardization effort called Next Generation Computer Resources of which the PSESWG is a part.. This program is designed to fulfill the Navy's need for standard computing resources while allowing it to take advantage of commercial products and investments and to field new technology advances more quickly and effectively.

The NGCR program revolves around the selection of standards in 6 interface areas.:

- Multisystem Interconnects:
  - Local Area Network 1 (SAFENET I)
  - Local Area Network 2 (SAFENET II)
  - High Performance Local Area Network
- Multiprocessor Interconnects:
  - Initial Backplane (Futurebus+)
  - High Performance Backplane
  - Switch Network
- Operating System
- Data Base Management System
- Programming Support Environment
- Graphics Language/Interface

Two of these areas are operating system interface standards and interface standards for project support environments. These interface standards are to be used to develop systems that are Ada-oriented, real-time, distributed/networked, multi-level secure, reliable and realizable on heterogeneous processors.

Application of these interface standards will change the Navy's approach from one of buying standard computers to one of procuring computing resources which satisfy the interfaces defined by the standards. These standards will be applied at the project level rather than a Navy-wide procurement level.

These interface standards will be based, to the greatest extent possible, on existing industry standards. In cases where existing industry standards do not meet Navy mission-critical needs, the approach is to further enhance the existing standards jointly with industry, thus assuring the most widely-accepted set of commercially-based interface standards possible.

The NGCR Operational Requirements describe some of the desired characteristics of the computer systems which can be procured using the new interface standards:

* A full-range family of computing resources, related through a set of interface standards, in a wide range of performance levels; software compatibility at appropriate levels is a necessary part of the "family" relationship
* Integration of multiple, dissimilar (heterogeneous) processors
* Internal and external standard interconnection; i.e., an internal computer interconnection (bus) to provide for growth in internal capability by configuring more modules and an external interface to provide for combining computing systems
* Incremental computing system growth; i.e., if a new function is needed, new modules or computers would be added to a system, and adding the new components would not require replacing the old system.

An objective of the NGCR program is to restructure the Navy's approach to acquisition of standard computing resources so as to take better advantage of commercial advances and investments. It is expected that this new approach will result in reduced production costs (through larger quantity buys), reduced operation and maintenance costs, avoidance of replication of Navy RDT&E costs (for separate projects to develop similar computers), and more effective system integration.

The Operating System Standards Working Group (OSSWG) has been meeting for 16 months with the majority of its members from industry and academia. The OSSWG has selected the IEEE 1003 (Posix) family of standards as the base-line for the operating system standard. Note that this baseline is for the OS to be used in the target (embedded) systems, i.e., the real-time OS in Navy missiles, ships and aircraft. The PSESWG is not at this point committed to using the same OS standards as those chosen by OSSWG or even to including any OS standards as part of the PSESWG set of standards.

C Development on PCTE
Ian Simmonds, SFGL
27-9-90

1. Introduction

Over the past five or six years a number of European companies have been working towards the production of a *Public Tools Interface* (*PTI*). The goal is to provide a powerful framework to support the next generation of software engineering environments (*SEEs*).

A vital part of the resulting *PCTE* (*Portable Common Tools Environment*) PTI is an object management system (*OMS*) which, while adding powerful data modelling facilities and integrity controls, can be seen as an evolution of the Unix filestore. Indeed, Unix has been a consistent source of inspiration throughout the PCTE project and other related projects.

This paper seeks to demonstrate some of the facilities of PCTE's OMS by showing some aspects of how a simple C development project might be organised.

2. Viewing PCTE as an enrichment of Unix

The PCTE OMS can be seen to be a filestore, with *objects* in the object base corresponding closely to *files* and *directories* in a filestore. A number of Unix concepts are replaced or extended in PCTE. In particular:

+ the object base is no longer strictly hierarchical. Any pair of objects can potentially be connected by a pair of mutually inverse links.

  This allows a large amount of structural information to be stored in PCTE terms, instead of being maintained by each Unix tool in a tool-specific manner, as well as allowing the user the possibility of reaching the same object by navigating along many different paths. The *look and feel* of navigating in the OMS is precisely that of following the directory structure of Unix, except the structure is no longer hierarchical! Existing PCTE command line interpretors (shells) even have built commands similar to *cd*, which allow the *current working object* to be changed by following a named link leaving the current object. Link names can even be concatenated to form paths. However, in this *graph structured* world, objects no longer appear at a unique path, and so the Unix notion of *path* = *name* = *pathname* no longer applies.

+ all objects and links have a *type*. This type is used to enforce a meaningful (to users *and tools*) organisation of the object base, as will be seen later.

  For example, a *C_Directory* can have links of type *.c* leading to object of type *C_Source*, and links of type *.h* leading to objects of type *Object_Code*. Further examples will be seen later.

+ all objects and links may carry attributes. The set of attributes available on a given object or link depends on the type of object or link.

  For example, a *C_Source* file may have an attribute listing all of the C compiler *-D* options which would influence the compilation of the code. A *C_Directory* may have a boolean attribute *Debug* saying whether or not all of the source code in the directory should be compiled with the C compiler's *-g* option.

+ object types form a type hierarchy, with a subtype inheriting all link types and attributes available for its parent type (recursively) as well as itself being extended with new properties.

  For example, we can imagine two subtypes of *C_Directory*: *Archive_Directory* designed to contain the source code and include files of an archive file (library); and *Tool_Directory* designed to contain the source code of a tool.

3. Introduction to the scenario

Let us now start building up a scenario to illustrate some aspects of C development on PCTE. At the moment we will start thinking about a programmer who is developing a simple X windows/Unix application as part of a statistical package. His chosen architecture consists of two windows based on his own widget set. The application is to allow the entry of data for a particular statistical analysis in one window, and will then display of results in a second window. The underlying statistical package is available as a C library. He decides to structure his software into a number of libraries, some of which are already available in his environment. The list of libraries is as follows:

C the standard C library - which everything calls

M the standard maths library

X the latest X windows library

S the library of the third party statistical package

D the programmer's widget set ('D' stands for 'Dialogue')

W1 a library containing all of the code for the first window

W2 a library containing all of the code for the second window

There is also a small amount of high level code associated with the initialisation of the tool, which is kept as a separate module. See Figure 1 for how this might be organised in the OMS. Notice the similarity to the Unix organisation. All objects shown here can be considered to be directories, although of several different types.

The programmer has been extremely careful to break his code up into libraries, because he wants to make use of the EAST build tool. This is a tool which knows quite a lot about how to build C code. Moreover, it doesn't need makefiles - it learns about the structure of an application from the way the user organises his files into archives. We will see various aspects of the EAST *C Builders* - the customisation of the EAST Build tool for the C language - as we continue.

4. Facilities for simple compilation

A *C_Directory*, as mentioned earlier, is a directory that contains the C source files and include files of a module. It can also store object code and a number of other types of thing required to support all Unix/C facilities (lint, assembler, ...). Figure 2 shows a typical *C_Directory*, containing various objects. Naming conventions have been intentionally kept the same as in Unix.

An EAST Builder is a tool which knows all of the rules for building within a particular type of *build context*. A *build context* is just an object playing the role of a directory, and any *object type* can be consider as a *build context type*.

The C builder associated with the *C_Directory* type knows that when it is applied within a *C_Directory* it has to produce an *Object_Code* file for each *C_Source* file within the *C_Directory*. In Figure 2 the tool has generated *a.o* from *a.c*, and *b.o* from *b.c*. Moreover, a number of attributes associated with the *C_Directory* object type can be set on each instance of this type to tell the C compiler how to compile for all source code within this *C_Directory*. These attributes correspond to C compiler options for *debug*, *optimisation*, *machine architecture*, ...

The C builder sets attributes on each object code file that it generates, recording what options were used to generate it. This information can be used to allow the automatic recompilation of individual source files when the *C_Directory*'s attributes are updated. Being a dedicated tool for building C code, a number of other checks are built into the *C_Directory* builder such as the comparisson of modification dates of the object code and the source code, and with

any included files.

The *C_Directory* type is not particularly meaningful in its own right, but acts as a parent type for two more meaningful subtypes. These are presented in the next two sections.

5. Archive directories

An *Archive_Directory,* as mentioned earlier, is a subtype of *C_Directory* which gathers together all of the source code and include files of a library (archive file). The include files of an *Archive_Directory* fall into two categories: those that are needed for the compilation of the source code of the library itself, and those that are needed by any consumer of the library.

The builder associated with the *Archive_Directory* type is a simple extension of that for a *C_Directory*, and merely takes the object code produced by the *C_Directory* builder and archives them in an object of type *Archive_File*. The *Archive_File* is linked to the *Archive_Directory* by a link called *.archive*. Each *Archive_Directory* contains the source of just one library, and each archive is named *.archive* with respect to its *Archive_Directory*.

One can imagine two views of an *Archive_Directory*. One view is for the developer of the archive, who needs to know about the source files used to develop the library, as well as any private include files needed to compile it. The other view is that of the consumer of the library. He is most interested in the archive file maintained by the *Archive_Directory*, and the include files needed for using the functions in the library. In fact, for third party libraries, it is unlikely that any source code will be available, and such *Archive_Directories* will look more like Figure 3.

The *Archive_Directory* type has a link type associated with it, called *This allows a consumer/consumed relationship to be maintained between Archive_Directories*. For example, the *window2* module within are example uses functions from the *stats* library, and functions from the *dialogue* library. The ANSI-C specification of the used function in the stats library is:

```
#include <tables.h>
void SetUpTable (TableT *Table, DataT *Data)
```

and in the dialogue library is:

```
#include <widgets.h>
void CreateButton (ButtonSpecT *ButtonSpec, CallBackT *CallBack)
```

Our programmer knows, from experience, that to use these libraries he must create *.ar* links to their *ArchiveDirectories* from *window2*'s *Archive_Directory*, as shown in Figure 4, and in his code for file *display.c* he puts:

```
#include "dialogue.ar/widgets.h"
#include "stats.ar/tables.h"
```

When we first thought of doing this we thought we would have to modify the C preprocessor so that it first looked for include files relative to the directory in which a source file was being compiled. There was no need to! This was already a functionality of standard *cpp*. It works really well for nested include files. See Figure 5 for a nice example!

It is expected that only two syntaxes are necessary for looking for include files. The first looks for include files directly within the local *C_Directory/Archive_Directory* and has syntax:

```
#include "____.h"
```

and the other uses an include file from a directly referenced library, with syntax:

```
#include "____.ar/____.h"
```

The use of <____.h>, "____.ar/_____.ar/____.h" et al is not recommended. This seems to be consistent with our philosophy of good programming practise.

6. Tool directories

The *Tool_Directory* object type is remarkably similar to the *Archive_Directory* type, except that the result of its associated builder is to link the object code produced by the *C_Directory* builder and the archive files of any referenced *Archive_Directories* to produce an executable. The executable is generated in the *Tool_Directory* at the end of a link with name *.exec*. Links of type *.ar*, as mentioned in the previous section, allow the programmer to identify the libraries *directly called* by the code in the *Tool_Directory*.

The final structure of our application is shown in Figure 6. Source files, include files, archive files and executables are not shown, for reasons of clarity. The use of the C library, with its abundant include files, is also not shown, and is a great inconvenience. A later comment complains about the Unix organisation of include files and libraries.

The graph of *.ar* links shown in Figure 6 defines a partial ordering of libraries. This is used by the *Tool_Directory* builder to calculate a total ordering that is suitable for telling *ld* in what order the libraries must be linked.

7. Generic Build algorithm

So far we have talked about the building of various types of directories as if they were reasonably independent entities. In fact, there is a second layer to our build tool which coordinates the building of many related directories. This generic build tool understands that it should not call a builder within a build context:

i until all build contexts on which the build context depends (sub-build context) have been successfully built

ii if there has been any error whil building within a sub-build context.

In our case the dependencies between build contexts are clearly defined by the *.ar* links. Nothing can be built until the maths library has been successfully built. Window 1 cannot be built successfully if there has been an error in any of *D*, *X* or *M*. And so on...

It is very important to notice that a builder may look at a build context and decide that it is in a state where nothing more has to be built. This is especially true in the case of system libraries, but also holds for modules developed earlier in the project, or even modules that have not been modified since the last time the build tool was executed.

Note also that there is no need for the user to be explicitely aware of all archives that are needed for linking his tool. In our example, our coder should not really be aware of the X library or the maths library in order to code modules *W1*, *W2* and *T*. He is programming using functions defined in the interfaces of *D* and *S* which themselves refer to *X* and *M*, but this is actually an *implementation detail.*

8. Other work in the EAST/Atmosphere context

This scenario has been based upon single version, single configuration development. The addition of multiple versions of each module, and multiple configurations of versions of modules innevitably increases the complexity of the whole process. This increase in complexity is not great, but requires a great deal more explanation. The extensions involved in providing a full environment with good configuration managament are the adoption of the version management facilities of PACT/PCTE+; and the separation of source files from derived objects in a build context. Once this is done it is possible to maintain many sets of derived objects per build context, and to have many architectures based upon the same components.

All facilities of the EAST environment will be provided through a graphical, object oriented user interface. The EAST environment also features advanced task management and process modelling facilities, and flexible documentation support. It is an open environment, with its own high-level and well-documented PTI. It will be available mid-1991.

9. Challenges to the Unix world

The approach presented above poses a number of challenges, especially when trying to transfer the existing search-path dominated library and include file organisations of Unix into *Archive_Directories*. There are a lot of useful facilities in Unix, if only they were better organised...

Challenge 1

Can someone please fight through the Unix libraries and include files and put them neatly into *Archive_Directory* format, complete with *.ar* links.

We have tried to do this, but had to settle for a small subset of Unix. We will slowly add libraries and include files as we need them.

The C builders presented in this paper call existing Unix tools. To do this we have had to use quite a few hacks. Moreover, the information available in our data organisation would allow more powerful versions of the basic Unix/C tools, such as *ld*, *cc*, and *ar* to be built. Most of our PCTE/C tools are implemented as capsules which themselves call Unix tools. Ideally we would like to be able to write real PCTE tools based on the many *Unix-indpendent* parts of the Unix world. Any Unix tool that accesses a Unix file is potentially very difficult to encapsulate!

Challenge 2

Is there any possibility of features such as the parser and code generator of *cc*, or the link editor of *ld* being available as libraries in Unix, so that we can write PCTE tool versions of these tools in a more convenient way?

This paper has concentrated on the C programming language. It seems clear that the approach taken is highly relevant for other languages including C++ and Ada. We are doing some work to apply our ideas to other languages, but it is a large task.

Challenge 3

Think about applying our approach to your favourite programming language. We expect royalties...

10. Acknowledgements

Much of this work was started in the ESPRIT project 'PACT', in which ideas were thoroughly explored. Particularly useful in the PACT project were the configuration management working group, consisting of Karima Berrada, Regis Minot, Ian Thomas and myself. The ideas have since matured in the context of the ESPRIT project 'ATMOSPHERE' in which Karima, Regis and myself have been joined by Francoise Lopez, Jean-Marc Morel and Chantal Vivier.

11. References

In the next version...

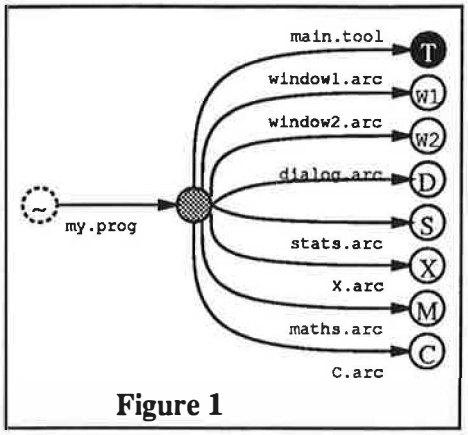


Figure 1

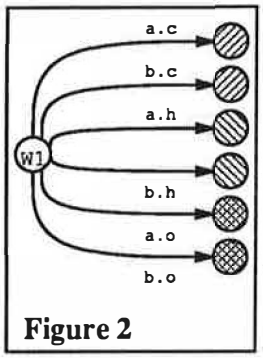


Figure 2

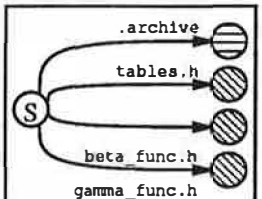


Figure 3

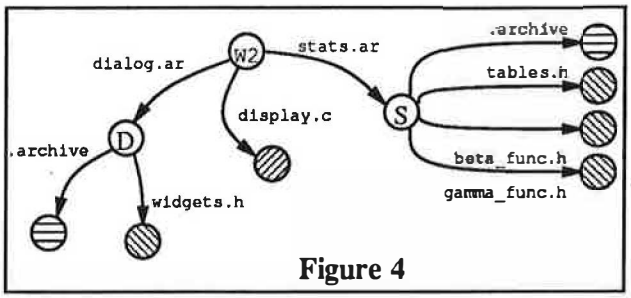


Figure 4

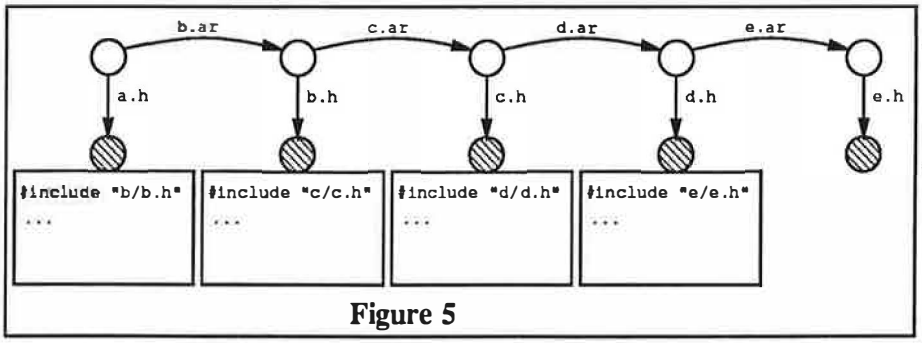


Figure 5

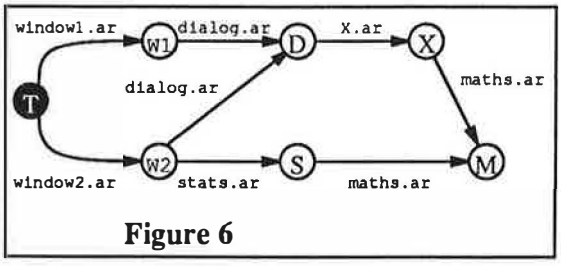


Figure 6

Supporting Parallel Programming Environments with Shared Persistent Data Structures

David C. Sowell
dcs@cc.gatech.edu

Karsten Schwan
schwan@cc.gatech.edu

College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280

1 Introduction

All software development environments (SDEs) are composed of a number of tools communicating with and controlling each other. This is typically done by using various ad hoc methods: interprocess communication (IPC) mechanisms for transient data, files for long-lived data, and invocations and interrupts for control. We are building several SDEs specialized for developing high-performance parallel programs. A database-like mechanism integrates our tools, providing a coherent framework for controlling each other and for sharing both short- and long-lived data structures. This mechanism, called Shared Persistent Data Structures (or SPDS, pronounced "spuds"), gives tool writers a way to create and manipulate persistent data structures in a manner very similar to the way they create and manipulate non-persistent ones. It supports concurrent and distributed access in an environment of networked workstations. Data are manipulated at full main memory speeds and except for the necessary locking operations to control concurrency, the programmer is not burdened with calls to an I/O subsystem; the persistent memory (disk) reads and writes and the distributed shared memory (DSM) cache coherency are handled invisibly.

We are currently working on two parallel computing projects, SPOCK and CHAOS. The SPOCK project[1] uses a custom distributed memory multicomputer, a "Parallel Function Processor" (PFP), for developing real-time simulations. This machine is hosted by a Sun386i. The CHAOS project addresses the development of operating system support for parallel machines. Its currently emphasizes highly dependable real-time systems and domain-specific programming system support. These are done on a BBN Butterfly, a shared-memory multicomputer running Mach.

2 An Example: The SPOCK Environment

A SPOCK Parallel Function Processor has 32 independent processing nodes which communicate through a crossbar network. The network configurations (connectivity and communication synchronization) are controlled by a network sequencer. Multiple PFPs are supported by a Sun386i

[1]This effort is in conjunction with the Computer Engineering Research Laboratory in the School of Electrical Engineering at Georgia Tech.

host. The typical way to program this hardware is to write separate program modules to run on various processor nodes and to write a specification of the communication patterns between those modules. To run the parallel program, the modules are compiled and loaded on the desired processors, the code generated from the communication specification is loaded into the sequencer and crossbar memories, the crossbar is enabled, and the processors and sequencer are started.

A PFP can support many different devices at its processing nodes. Current devices include Intel 286 and 386 boards, custom high-speed floating point and double-precision processor boards (FPPs and FPXs), analog-to-digital/digital-to-analog boards, and array interconnect boards for connecting multiple PFPs together. Because real time performance is of utmost importance, the hardware configuration is quickly and easily changed to trim communication costs for a particular program. Other programs, however, must be able to cope with a less-than-desirable configuration. Applications are insulated from node heterogeneity and unexpected hardware reconfigurations by the programming environment.

The SPOCK programming environment, called the Integrated Parallel Programming Framework (IPPF), consists of four primary tool sets:

1. The Parallel Program Construction System (PPCS) includes all tools used during program development: a graphical user interface, recompilation drivers, compilers, assemblers, program libraries, linkers, and loaders.
2. The Parallel Program Monitoring System (PPMS) are the performance monitoring and visualization tools. These tools are currently being designed based on experience with the CHAOS project.
3. The Parallel Program Tuning System (PPTS) currently consists of a compiler which produces process-to-processor mappings and interprocessor communication patterns. The PPTS will ultimately include a graphical user interface where a programmer can easily reconfigure and tune a parallel program for its target execution environment.
4. The Tool Integration System (TIS) provides the basis for the PPCS, the PPMS, and the PPTS to work together effectively and efficiently. It consists of several tools: a number of device drivers to control the PFP hardware, a library supporting host program interaction with the device drivers and the parallel programs running on the hardware, an automatic hardware configuration tool, and the PFP resource manager.

A programmer uses the PPCS for the initial development of a parallel application. When the program is complete, the programmer runs it and gathers performance data with the PPMS. Based on the performance measurements, the PPTS is used to tune the application for a particular execution environment. These three tool sets are supported by and coordinated by the TIS.

The IPPF tools use SPDS databases to store and share information and to coordinate themselves. The automatic hardware configuration tool creates a database to describe the hardware (e.g., node types and crossbar addresses) and to relate it to particular Unix device drivers (major/minor numbers.) The resource manager uses this information plus information in a software configuration database maintained by the PPCS and PPTS to reserve PFP hardware for a particular application. If the resources requested in the software configuration do not match the available hardware configuration, it invokes a tool to do on-the-fly software reconfiguration, changing the process to processor mapping and altering the communication pattern accordingly. The reconfigu-

ration may be as insignificant as swapping one processor for another of the same type, or potentially as difficult as automatically recompiling some modules to run on a different types of processors. Processes might even be remapped to different PFP arrays communicating with the rest of the application through array interconnect boards.

3 The Design, Implementation, and Evaluation of SPDS

Design: Typically, programmers use `malloc(3)` to dynamically allocate memory from the heap. Instead of the heap, SPDS allocates chunks from a regular Unix file and uses Unix's virtual memory facilities to map the chunks directly into main memory. Before manipulating a persistent data structure, a programmer specifies the name of the file where the structure will be stored. After the file is opened, he refers to the memory chunks with pointers. Several files can be used simultaneously, thus naturally providing a variety of name spaces or independent structures.

SPDS represents pointers to persistent memory in two ways. A single process can simply use regular pointers (i.e., virtual memory addresses.) This representation cannot be (usefully) shared with any other processes, however. The external pointer representation is simply the offset of a chunk from the beginning of its file. SPDS supplies two routines for translating between the two representations. Thus, complex dynamic data structures can be built with SPDS. In fact, above its basic functions, SPDS also provides construction and manipulation routines for persistent sets and symbol tables.[2]

Implementation: The current implementation of SPDS is based on the Sun Microsystems Network File System (NFS) and the System V record locking mechanism as provided by `fcntl(2V)`. When SPDS opens a database, it uses `mmap(2)` to map the file directly into the process's virtual memory. As new persistent objects are allocated, the file is extended in increments of the system's page size, and new pages are mapped into virtual memory accordingly. (SPDS is designed to work between machines with a variety of page sizes.)

Sharing is done at the page level. When a process needs access to a particular data object, it locks it using one of the SPDS locking calls. SPDS finds the pages spanned by the object and sets advisory locks for each page using `fcntl()`. If the pages are mapped from an NFS file, the pages are invalidated with `msync(2)`. The next reference to the object causes a page fault which causes NFS to get the current state of the page from the host it is mounted from. Similarly, the SPDS unlocking calls use `msync()` to copy the state of the page back to the remote mounted file system. Non-NFS files (i.e., files on a local disk), do not need to invalidate or flush pages, since all processes will use that host as a repository for the states of unlocked pages and all processes local to that host share the same pages of physical memory to which the file is mapped. Note that because sharing and locking are done at the page level (as is required by `msync()`), objects within the same writelocked page are only accessible to a single process at any one time, throughout the network. SPDS allows a single process to lock multiple objects within a page and it handles it intelligently. The page is invalidated only when the first lock is obtained and is flushed when the last writelock

[2] Non-persistent sets and symbol tables are also supported with the same routines. They can be used in conjunction with their persistent counterparts.

is released. The page is writelocked as long as one object within the page is writelocked. Otherwise it is readlocked.

Problem: SPDS currently makes heavy use of the `fcntl()` system call. Unfortunately, this mechanism's semantics do not match those needed by SPDS. It causes `mmap()` to fail when used with NFS files (in Sun's implementation, at least). It is also quite slow. SPDS needs a mechanism which supports true shared-memory locking operations. This would reduce the overhead for locking calls in many cases by several orders of magnitude.

The semantics of `fcntl()` do not guarantee that lock conversion is atomic: if a process has either a readlock or a writelock on an object and it wants to convert it to the other flavor, `fcntl()` does not prevent a second process from obtaining a writelock during the conversion and changing the state of the object. The current SPDS implementation requires atomic lock conversion. To support atomic conversion, SPDS associates two locks with each object. It uses the first one to provide exclusive access to the second. The second lock is used for the normal readlocks and writelocks for the object. SPDS obtains the first lock before attempting to convert a readlock to a writelock or *vice-versa*. It releases the first lock immediately after the conversion attempt. Thus, each atomically-convertible object lock involves three `fcntl()` system calls.

On a 25 MHz Sun386i, each `fcntl()` call takes about 11 ms, independent of the type of lock being set (readlock, writelock, or unlock) and whether or not the lock was granted. Each call generates network traffic even if the lock pertains to a file on a local disk and even if the workstation is not actually on a network. Because object locks must be implemented with three `fcntl()` calls, they take more than 30 ms each. The performance of `fcntl()` is an order of magnitude better on SparcStations, but we feel that this is still too slow. Locks for objects on a local disk should be denied in less than 10 μs (only a few instructions.) They should be granted in less than 100 μs, depending on how much bookkeeping has to be associated with granting the lock. Locks for objects on non-local disks could potentially be denied or granted equally as quickly, depending on if the object is cached and how the DSM mechanism is implemented (whether sharing is done at or above the object level.) If the object is not cached, the lock will have to be accessed across the network, incurring the associated network delays.

Solution: We are currently involved in redesigning the locking mechanisms used by SPDS. Initial measurements have indicated a two to three order of magnitude performance improvement for locking objects on a local disk. We have noticed another anomaly with `fcntl()` locks: when more than twelve SPDS-based processes are running concurrently on a single machine, their combined execution times begin to diverge radically from the combined execution times of their serial execution. This is not true of our redesigned locks. At twenty-three processes, the `fcntl()` versions run concurrently at 733% of their combined serial time while the new versions run at 88% (they actually run *faster* concurrently than serially – we think this is due to paging overhead being shared between the concurrent versions). We have not done a complete analysis, but we think that the divergence of the `fcntl()` versions is at least partially due to the double-level locking required for atomic lock conversion.

Integrated Project Support Environments: Benefits and Functions

Walter Van Riel
Atherton Technology
1333 Bordeaux Drive
Sunnyvale, CA 94089
Walter@ATHERTON.COM

Abstract—Software BackPlane, the integrated project support environment (IPSE) provided by Atherton Technology is presented in this paper. Software BackPlane enables process engineers to model and automate their organizations' software development process. The major functions provided by the Software BackPlane IPSE are: persistence by means of an object-oriented database; a typing capability; versioning and configuration management; audit trails; browsers; and extensibility. The extensibility of the system permits third party software tools to be integrated and allows the flow of data to be modeled, i.e., workflow control.

INTRODUCTION

The term *integrated project support environment*, also known by its acronym *IPSE*, has already become a common buzz word in the world of software engineering. But what exactly is an IPSE, and of what value is it? This paper explains the benefits of IPSE technology and discuss how IPSE's can be applied to problems in software development organizations.

This paper focuses on Software BackPlane, the IPSE offered by Atherton Technology. Software BackPlane is one of the few American IPSE products commercially available today. Software BackPlane provides a framework with built-in facilities that enables software development organizations to build and model their own environment.

This paper explains the concept of an IPSE by describing the basic features of Software BackPlane and how they relate to real problems.

The Value of IPSE's

In working with our customers, we have visited many different software development organizations over the past few years. Even though each software development organization has its own unique set of development needs, we have observed that there is much commonality in the software development process.

A number of companies have attempted to improve their software development processes. Typically these organizations start by creating the base of functionality (such details as building multi-user object-oriented databases, implementing generic versioning and configuration management schemes, worrying about extensibility, portability, user interfaces, and so forth) on top of the raw file system and its tools. With an IPSE technology, the basic building blocks for modelling and automating the development process are already in place, with a minimal effort and investment and without having to reinvent the wheel. Process engineers, the builders of those custom environments, can concentrate on solving the problem at hand.

A framework that provides built-in tools is not necessarily a new idea. Historically, MIS departments provided automation by writing programs to transfer data into and out of native files; this process has been replaced, with the advent of commercial, off-the-shelf relational databases. Now it is a widely accepted practice to store data in these databases and use simple programs or high-level query languages to store and retrieve the data. IPSE's have the potential of becoming to software development organizations what relational databases have become to MIS departments. In fact, we may someday see companies that do nothing but build various custom software development environments.

IMPORTANT FUNCTIONS OF AN IPSE

A good way to understand the value of an IPSE is to take a look at the basic features of Software BackPlane and how they relate to real problems in a software organization.

Persistence Using Database Technology

Modeling a software development environment requires storing data about the real life objects in the process, including the semantic operations that can be performed on that data, that is, persistence.

While some engineers advocate using raw file systems to store this information, others believe that databases are more appropriate. The debate of using

file system versus database technology will remain open for the foreseeable future. We at Atherton provide persistence, the ability to store data and related semantics by means of a proprietary object-oriented database.

Objects modeled in the database represent entities such as users, groups of users, and roles that users can assume in the organization. Other objects represent instances and definitions of object types such as software code, various types of documents, action requests, test reports, etc., and collections of the above configured into logical units.

Since an object-oriented database approach was chosen, every object that can be modeled is typed. The set of all types in the database represents collection of objects, the logical schema of the database. This schema is very useful for performing queries and operations (methods) related to instances of those types.

As an example, let us assume that an organization wants to establish a type definition for a collection of objects (similar to a UNIX™ directory) to control the information that can be held in that collection. Let us define a new type of collection called "SUBSYSTEM" that contains such things as source code, header files, a Makefile, etc. Let us also assume that there is a requirement that each instance of a SUBSYSTEM collection contain a test suite. We can define another new type to represent this test suite called "SUBSYSTEM_TEST". We can specify that this test suite contain only test cases that apply to the associated subsystem.

This is a relatively simple task in a typed system that uses database technology. In this example, the user can query all the instances of type "SUBSYSTEM" or "SUBSYSTEM_TEST" and retrieve the associated test suite for a subsystem. The types can be defined to prevent users from putting test cases in the subsystem collection or from putting source into the test suite collection. Performing queries like these and enforcing such rules is hard to accomplish on top of a raw file system. Typically, naming and placement conventions in a raw file system must be employed to provide similar capabilities.

Another benefit derived from using an object-oriented database technology is that operations are encapsulated around transactions. This allows the user to define single atomic operations that succeed or fail as a whole, without the danger of leaving any data in an inconsistent state in case something goes wrong such as running out of memory, access conflict with another user, power failure, etc. It also guarantees that the data is always in a consistent state.

Of course, databases must support many users and ensure that access conflicts are avoided when users try to read or write the same object simultaneously. It is also important to have sufficient database performance and capacity to be of practical use.

Typing and Type Hierarchy

An Atherton database contains by default an initial type hierarchy as shown in Figure 1.

The schema is defined in terms of itself, that is, the types descended from ELEMENT are instances of the type called ELEMENTTYPE, and the datatypes (descended from VALUE) are instances of the type DATATYPE). Instances of ELEMENTTYPES define attributes (also called instance variables) for that type; attributes can be of various types, such as strings, integers, date time stamps, and links to other objects in the database. The links that are currently supported are one to one, one to many, and many to many.

Link attributes are represented in the type hierarchy by the datatypes ELEMENTID and SCAN and are of specific value. They provide the building blocks for linking the various types of objects in a software development environment with one another, such as linking code to its documentation, test suites to the source it is supposed to test, problem reports to the version of the product where they have been identified and fixed, etc. Providing this type of capability on top of the raw file system level is particularly hard to do.

Like any object-oriented system, there is a set of messages that can be invoked on instances of any given type. Messages and instance variables can be inherited. In Software BackPlane, only single inheritance is currently supported.

Methods bind messages to type and are represented as the matrix elements shown in Figure 2. The matrix shows only a few types and a few messages, but shows how methods bind these messages to the types. For example, a method exists that implements the *reserve* message on the VERSION type. Correspondingly, there is no method to implement the *open* message for VERSION. Software BackPlane provides inheritance to make the task of defining methods easier. You can define a new method or have a method refine or inherit behavior from its supertype.

Since objects are modeled at a high level of abstraction, the cost of run-time binding, i.e., locating the method to be executed when a message is sent to an instance of a type, is of less importance than in the case of those object-oriented systems that support programming in the small such as Objective C™ and SMALLTALK-80.

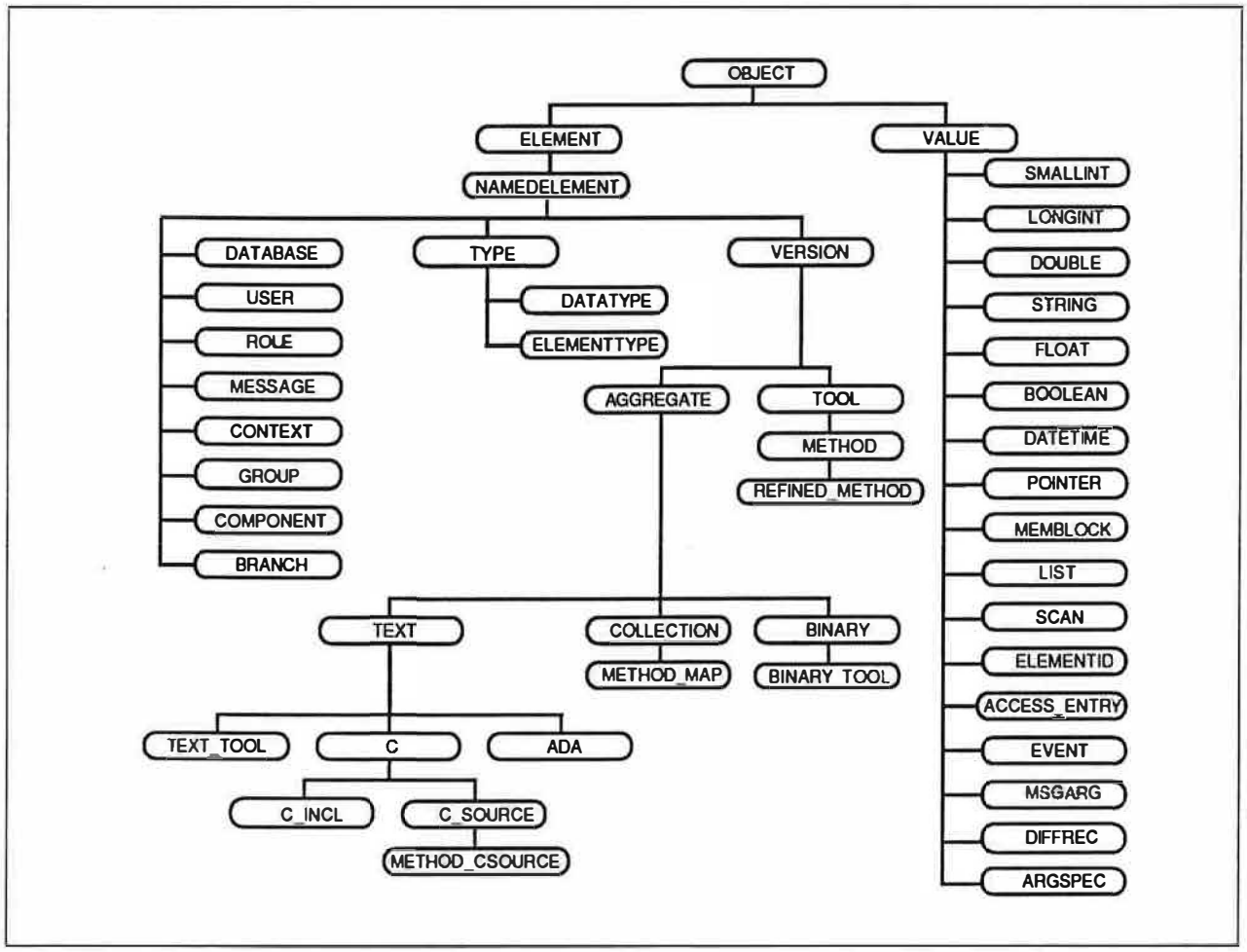


Figure 1 The Type Hierarchy

TYPES –> *messages*	VERSION	TEXT	C_SOURCE	COLLECTION
reserve	√	√	√	√
replace	√	√	√	√
open		√	√	√
compile			√	

Figure 2 Relations between Types and Messages

In contrast to most object-oriented systems, Atherton chose to use methods to bind messages to types according to the user's access permissions and role in the software development organization. Note that the bindings in Figure 2 can be completely different for different user/role combinations.

As an example of binding, a user in the role of "development engineer" can be enabled to send a *release* message to an instance of type SUBSYSTEM, in order to send an authorized version of the code to the release engineering group after some standard checks are performed. On the other hand, the same user while in the role of "release engineer" may not be permitted to perform this action, but can have different messages available such as *buildReleaseTape*.

Tools that are available in the software organization are invoked as a result of executing a method. For example, the *open* message for a document written in FrameMaker™ would bring up the FrameMaker product on that data. That same message invoked on a module written in C can bring up the emacs or vi editor, based on the user's preference. The message *compile* on a C source object can invoke the compiler or cross compiler. Thus, we use the methods in the IPSE to invoke the integrated, encapsulated tools. The tools themselves can be stored as versioned objects in the database or can be kept outside, based on the system modeler's preferences.

Messages are the building blocks of the IPSE in the same way that LEGO™ blocks are used by children to make different kinds of toys. Thus, the more functional reusable building blocks provided by the IPSE vendor the easier will be the task of the process engineer.

As an example of reuse, the *archive* message for the SUBSYSTEM type can be used in the release procedure, but can also be used in the operation that produced a product tape. This reuse capability is one of the important factors behind the popularity of object-oriented technology.

Another important aspect of an object-oriented database is the ability to extend the logical schema of the database (i.e. adding types, messages, and methods) on the fly, without invalidating preexisting objects in the database or undergoing difficult database conversions. In Software BackPlane, we made it very easy to extend the schema, and when properties are added to a type of which instances already exist in the database, an error "property not set" will be generated if an attempt is made to query that property on those instances. This works fine if method code is written to manage this case.

There are many additional benefits from typing data. One can perform queries, such as "Where is the test suite that corresponds to this subsystem?", or "What has changed between this version of the product and the previous version?". Without a logical schema it is very difficult to construct these types of queries.

Another benefit from typing is that data can be interchanged from one format to another. For example, sending a message *readFrom* to an instance of a document written in FrameMaker after selecting a document written in TROFF may automatically invoke the troff to FrameMaker converter. Sending that same message by pointing to say an document written in Interleaf could invoke the Interleaf to FrameMaker converter.

It would be nice if typing capabilities came standard with operating systems such as UNIX. Since this is not the case, IPSE vendors have been forced to provide this functionality as database technology on top of the standard OS. Considering the installed base of UNIX systems today, it is not likely that typing will become an integral part of UNIX. Furthermore, trying to provide this functionality by requiring modifications to the standard OS kernel by a third party vendor would make end users feel uneasy.

Versioning

The statement "... but this code worked fine just yesterday" is something we all are unfortunately too familiar with. Probably one of the most difficult problems facing software development organizations today is managing change.

Version control need not be limited to versioning text files, as performed by UNIX tools such as RCS or SCCS — it is much more useful to version whole directories. Today, systems are delivered consisting of hundreds if not thousands of parts. A typical software product can have source code; graphic documents produced by design tools like IDE's Software Through Pictures™ or Cadre's TeamWork™; word processing documents produced by technical publishing tools such as FrameMaker or Interleaf; bug lists; various reports; product manuals; etc. Although some tools provide built-in versioning schemes, the real goal is to produce a single consistent version of an entire system. Managing different incompatible versioning schemes adds to the problems of change rather than alleviating them. What is really needed is a uniform versioning scheme that applies to all of the various parts that make up a system.

In Software BackPlane, we provide a standard type called COLLECTION which has the equivalent

functionality of a UNIX directory but can also be versioned. The type COLLECTION inherits its versioning capability from a supertype called VERSION and by definition can only contain versioned objects. Versioned objects can perform standard versioning operations such as *checkout* (reserve), *checkin* (replace), *cancel* (unreserve), *branch*, and *merge*.

As mentioned earlier, Software BackPlane is an object-oriented system. In Software BackPlane, new object types can be defined to be a subtype of VERSION. You can define appropriate subtypes of VERSION such as design objects, test object, source, shell scripts, etc. so that they become automatically versionable. Thus, you can preserve a directory of type COLLECTION containing thousands of individually versioned objects, all of which will be fully reproducible.

To provide this uniform versioning system for tools that normally operate on data in the native file system requires tool integration. Integrating a tool means ensuring that the operations such a *checkout*, *checkin*, and other related operations work as intended for the data of that tool. When a new versioned type is introduced in the database, the versioning methods may have to be refined, although sometimes they can be fully inherited. We will discuss a bit later how some tools lend themselves better to integration than others based on their architecture.

Audit Trail

When a message is sent in Software BackPlane, the method invoked can be defined to automatically generate a history record. This record is kept with the object that receives the message; it records what happened, when, why, and by whom. The user of the IPSE has full control to enable these audit trails on existing methods as well as user provided ones. This provides answers to such questions as : "Who changed this module?", "Why was this module reserved?", or "Who resolved this action request?". A good IPSE needs to provide such audit trailing capability and must make it easy for customization by end users.

Configuration Management

Configuration management is a term that has different meaning to different people. In this paper, configuration management is understood as the ability to version hierarchies of versioned objects allowing any member of that hierarchy to be of any type. This enables such hierarchies to become fully reproducible.

Software BackPlane also provides a capability called branching used to create a working copy of a specified version without altering the original version. Another capability called merging enables branched versions to replace the originals. The combination of the branching, merging, and versioning capabilities allows multiple people to make changes to the entire system without having to be affected by each other's changes. This is also known as parallel development.

In addition to parallel development in a multi-user environment, the branching capability can benefit individual users maintaining different branches of the same system. For example, a software engineer might have to support a product for multiple target machines. Managing the changes to that system for the various targets could easily become overwhelming without an orderly means of branching.

Access Control

When building a system to be used by multiple people, access control becomes an issue. It may be necessary to prevent some people from performing certain operations (e.g. *read*, *write*, *execute*, etc.) on other people's data.

In Software BackPlane, every object has an access control list (ACL). This access control list specifies the access privilege (e.g., read, write, control, execute, version, etc.) to an object according to qualifications set by user and/or group and roles. For example, one could specify that only the "database administrator" has the right to extend the logical schema. Or similarly, access control could be set up so that only the owner of an object has the right to change it. Another example would be to only allow users in the role of "quality assurance" to mark action requests (e.g. bugs) to be resolved.

Software BackPlane allows you to specify a default initial access for each newly created object. In addition, every message in the logical schema can specify the access required to perform that message against the receiving object. If there is not sufficient privilege for that object to perform the message, the system will generate an access violation condition.

Access control can also be used to limit the objects a user can view according to his role or group. For example, why bother to display the type "C_SOURCE" in an object creation menu to a user in the "technical writer" role if that role has no need to create C code? Thus, access control can also be used to limit the scope of objects a user interacts with.

Browsers

A major function of an IPSE is to provide convenient access to the data stored in the database. In conventional operating systems such as a UNIX window running the C shell or Bourne shell, one views project data hierarchically. In Software BackPlane, a type called CONTEXT corresponds in functionality to a UNIX shell, with the difference that its instances are persistent and that the home and current working directories are versioned collections. Contexts also support *context variables*, which are functionally equivalent to UNIX environment variables, with the additional functionality that they are persistent, and they can be associated with a version in the collection hierarchy.

Software BackPlane permits other views into that same data as well, in the same way the hypercard™ stacks allow you to traverse the same information differently. For example, it is possible to view the data based on type (e.g., view all instances of type C_SOURCE or MAKEFILE) as opposed to a hierarchical view.

The user interface to the IPSE data should support a command-based interface in addition to graphical interfaces, even though we are living today in the age of bitmap terminals. The convenience of being able to write command scripts and run them in batch mode should not be overlooked. It only takes one experience of having to perform the same sequence of clicks and clacks twice to achieve a similar end result to appreciate the value of a command-based interface.

Extensibility

The ability to extend the system is in the author's opinion the most important function of an IPSE. Extensibility translates into extending the logical schema by refining or defining new types, messages, and methods, and by setting up access control, defining groups, users, and roles. Being extensible requires the interface and schema of the IPSE to be public, both for read and write. The object-oriented model lends itself very well to extensibility because of its inheritance capability.

At one point in Atherton's history, an attempt was made to deliver an IPSE on top of the Entity Relationship model (ER). This proved to be very unsuccessful primarily because of the difficulty of extending such a system. Extending the ER model implied defining new domains, and defining new relationship between those domains. Functions that operated on the entities in those domains had to be packaged into procedural libraries. Every time new domains were defined to extend the system, the number of public functions increased, which quickly grew into an unacceptably large number of functions.

With an object-oriented model, most of the work is accomplished by only two interface functions: (1) the function that sends a message to an instance and (2) the function that sends a message to the supertype of the current type. New capabilities are added by extending the logical schema, not by adding to the set of external callable interface functions. Although some utility functions such as opening and closing the database and starting transactions are still needed, the use of the object-oriented model versus the ER model reduces the number of functions by more than an order of magnitude.

The way in which semantic values are provided to the IPSE can make a big difference to the productivity of the process engineer. Software BackPlane allows methods to be provided as functions written in the programming language C, shell scripts, or as scripts written in Atherton's proprietary interpretive language.

If semantics must be added to provide significant interaction with the database, it is preferable to write the methods in the language C. Methods that apply to the same type can be packaged in a single module which will be loaded dynamically when one of its methods get executed as a result of a *send* message. An example of such a method would be a new message on the type ELEMENTTYPE that returns the recursive list of all subtypes of that type. Since there is overhead in dynamically loading code, the IPSE vendor should allow the standard interface programs such as the browsers to be relinked with these newly precompiled methods.

Many UNIX operating system users have experienced the benefit provided by shell scripts. This need not be lost in an IPSE. Software BackPlane allows methods to be specified as shell scripts that will be invoked as a result of a *send* message. Arguments to the script are provided based on the arguments of the message and data in the data base. For example, it is possible to refine the checkout of the type C_SOURCE to invoke a shell script that will initialize the instances of that type when it is first checked out. This is accomplished by running a UNIX stream editor script over a C_SOURCE template, with both the script and template stored in the database. This makes it convenient to have programmers comply with module standards, such as copyright statements and module headers.

Finally, there is the hybrid of the two; a string-based language that can perform database operations in an interpreted fashion when a method is executed. An example of such a method could automate the creation of a new subsystem. It could instantiate the

initial parts of the subsystem: its Makefile; public and private header files; error files; an empty, preinitialized test suite collection; etc.

Extensions that have been added to the logical schema must become visible at the user interface level. For example, if a new *compile* message is defined on type C_SOURCE and ADA, that message should become visible in a pop-up menu when the user selects an instance of those types.

Tool Integration

When the database approach for providing IPSE functionality is chosen, a problem known as tool integration needs to be solved. Most commercially available third party tools that contribute to the construction of quality software are built to operate on data stored in files in the native file system, not on data stored in a proprietary database. The process of making these tools operate within the context of a database is known as tool integration.

Figure 3 shows an example of extensions made to the type hierarchy as a result of integrating the Interleaf technical publishing tool.

Normally, when using the Interleaf tool in the native OS, the user creates a desktop for organizing technical publishing work, e.g., books and documents. The work can be stored in folders, drawers, and cabinets. Books represent collection of documents, such as chapters, front matter, a table of contents, and maybe an index. By defining types that model the data of this tool, these objects can become versioned objects in the same way other objects in the IPSE are versioned. The data of the tool is still kept in the file system, but access to the data now is under control of the interface to the IPSE. Sending the *open* message to an instance of these types will bring up the tool on the data as expected.

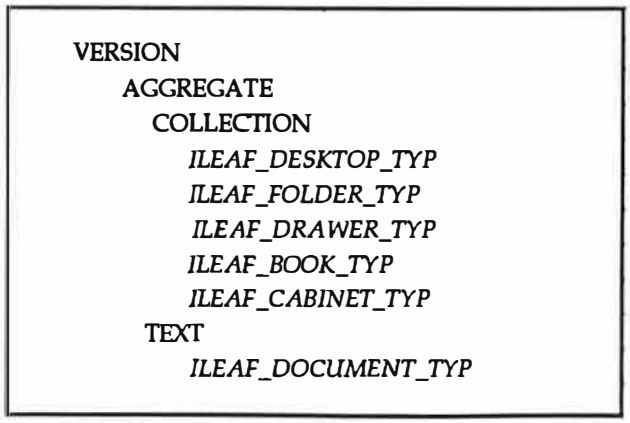


Figure 3 Type Hierarchy Example

For an IPSE to become fully functional, some third party tool integrations should be provided commercially. The IPSE vendor can provide these integrations. Experience has shown that the quality of an integration improves when it is performed by the tool vendor in cooperation with the IPSE vendor. Since the IPSE can be populated by diverse third party tools, it is effectively independent of methodologies, tools, or languages.

Integrated tools should cover the spectrum of software development: design and analysis, project management, desktop publishing, and other functions such as testing, compilation, integration system building, metrics gathering, and so forth. Of course, some tools lend themselves better to integration than others. Tools that allow the invocation of the tool and a procedural or command based access to its data to be controlled are easiest to integrate.

A concept useful in tool integration that we use at Atherton is to allow a native file system directory to be associated with an instance of a versionable collection. This creates a read-only file in this directory for all the immutable (checked-in) versioned parts of the collection. Checking these parts out creates a writable version in that directory. If such a scheme is employed, one can think of a the Software BackPlane as a collection of RCS or SCCS directories whose elements are encapsulated into an object-oriented database. The directories associated with these versioned collections are practically identical in function to the directories that are typically associated with the RCS or SCCS directories when those tools are used in stand-alone mode. When collection directories are used, the tools can be invoked from within the database or by invoking the tool directly on its data in these collection directories.

Workflow Control

Once the individual tools are integrated, the process engineer can make use of the subtyping to provide further functionality. To continue the Interleaf example above, a new type called IRD (Interface Requirements Document) can be defined as a subtype of ILEAF_BOOK_TYP. Adding new messages or refining existing ones can be used to enforce compliance with standards and link the requirements in the document to the code that implements the various requirements.

Similarly, messages can be added to automate the flow of data between separately integrated tools. A good example of this is automating the data flow from a project scheduling tool to a structured communication tool.

Role	Daily Tasks	Message/Type
Tech writer	Write release notes.	Message on type ELEMENTTYPE AR (Action Request) that will create a document in certain format that lists all the outstanding problems reported on a released product.
Software engineer	Release a subsystem.	A message to a collection of type SUBSYSTEM that will ensure the subsystem is releasable, i.e. compilable, without lint errors, ensuring all parts are checked in. Will inform by native mail to the Release Engineering group where to pick up the newly released subsystem.
QA engineer	Verify bugs.	Find all the action requests (query message to type AR) that have been reported resolved by engineers on a released system, and verify that they indeed are fixed.
Engineering manager	Prioritize tasks.	Create a report listing all outstanding tasks on a system, by priority (query message to type AR).
Release engineer	Make product tape.	Message *buildTape* on type PRODUCT, where PRODUCT is the collection that holds all the product's subsystems.
Program manager	Update product schedule.	Send a message to type PLAN that will find all action items resolved by the structured communications tool and update the plan accordingly.

Figure 4 Associating Roles with Messages

Action requests (i.e., a task to be performed by some project member) in the project schedule can be assigned and tracked by the communication tool. When an action request (AR) is marked say completed by its owner, the project schedule ought to be updated accordingly. This flow of data could be performed automatically. The process of automating the flow of data between different types of tools is an example of workflow control.

One way to define the workflow of an organization is to determine the roles performed by various people in that organization and their associated activities. This information can then be used to create formal definitions of roles, types, and messages to be implemented in the IPSE. Figure 4 shows roles and their tasks in certain roles and how they can be implemented into IPSE messages and types.

CONCLUSION

We have explored some of the exciting aspects of the IPSE technology. For this technology to prosper, it is the responsibility of the hardware, tool, and IPSE vendors to work closely together to deliver a complete and extensible solution to their customer base. It is the responsibility of those customers however to accept and commit to this technology, and exert pressure on the vendors to work together to provide a coherent solution that solves the software development problems rather than contributes to it.

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *;login:*; produces a quarterly technical journal, *Computing Systems*; co-publishes books with The MIT Press; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

Computing Systems, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and in addition, the Association will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX users and system maintainers who convened periodically to discuss problems and exchange ideas concerning UNIX. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the fees paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association
Suite 215
2560 Ninth Street
Berkeley, CA 94710
Telephone: 415/528-8649
Email: office@usenix.org
Fax: 415/548-5738

USENIX SUPPORTING MEMBERS

Aerospace Corporation
AT&T Information Systems
Digital Equipment Corporation
Frame Technology, Inc.
Quality Micro Systems
mt Xinu
Open Systems Foundation
Sun Microsystems, Inc.
Sybase, Inc.

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710